RDA-TR-43-0005-001

12

Final Technical Report

MJO: 0043

Neural Network Studies

Gregg Wilensky, Narbik Manukian, Joseph Neuhaus, Natalie Rivetti

Logicon RDA
July 1993

ARPA Order Number: 7006
Contract Number: N00014-89-C-0257
Contract Effective Date: 15 September 1989
Contract Expiration Date: 31 July 1993
Principal Investigator: Gregg Wilensky
(310) 645-1122, Extension 369

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Advanced Research Projects Agency or the U.S. Government.

LOGICON

RDA

LOGICON RDA ▪ POST OFFICE BOX 92500 ▪ LOS ANGELES, CALIFORNIA 90009

6053 WEST CENTURY BOULEVARD ▪ LOS ANGELES, CA 90045 ▪ TELEPHONE: 310 645-1122

RDA-TR-43-0005-001

Final Technical Report

MJO: 0043

Neural Network Studies

Gregg Wilensky, Narbik Manukian, Joseph Neuhaus, Natalie Rivetti

Logicon RDA
July 1993

ARPA Order Number: 7006
Contract Number: N00014-89-C-0257
Contract Effective Date: 15 September 1989
Contract Expiration Date: 31 July 1993
Principal Investigator: Gregg Wilensky
(310) 645-1122, Extension 369

LOGICON
RDA

LOGICON RDA ▪ POST OFFICE BOX 92500 ▪ LOS ANGELES, CALIFORNIA 90009
6053 WEST CENTURY BOULEVARD ▪ LOS ANGELES, CA 90045 ▪ TELEPHONE: 310 645-1122

Forward

This report was produced for the Office of Naval Research (ONR) and the Advanced Research Projects Agency (ARPA) in fulfillment of the final report requirements under Contract Number N00014-89-C-0257, Section B, Item 0002 and Exhibit A (Basic Contract) , Item 0004 and Exhibit B (Option I), and Item 0006 and Exhibit C (Option II).

Table of Contents

Introduction and Summary

Research at Logicon RDA in neural networks under the ARPA contract includes three main areas: theoretical research in the fundamentals of neural networks, applications of neural networks to practical problems of interest, and the development of new and more powerful neural networks and techniques for solving these problems. The papers which follow are a compendium of our research in these areas.

The theoretical studies include an overview of the basic useful theorems and general rules which apply to neural networks (in *"Overview of Neural Network Theory"*), studies of training time as the network is scaled to larger dimensions (in *"Scaling of Back-Propagation Training Time to Large Dimensions"*), an analysis of the classification and function fitting capability of neural networks (in *"Classification and Function Fitting as a Function of Scale and Complexity"*), a comparison of Bayes and k-nearest neighbor classifiers to neural networks (in *"A Comparison of Classifiers: The Neural Network, Bayes-Gaussian, and k-Nearest Neighbor Classifiers"*), an analysis of fuzzy logic and its relationship to neural network (in *"Fuzzy Logic and the Relation to Neural Networks"*), an analysis of the Reduced Coulomb Energy (RCE) network (in *"The Reduced Coulomb Energy Classifier"*), radial basis functions (in *"Radial Basis Approximations"*), and the Lynch-Granger models (in *"Lynch-Granger Model of Olfactory Cortex"*).

The application of neural networks focuses on practical problems such as the prediction of the outcome in the growth of solar cells as a function of the controlling parameters of the process (in *"Multi-parameter Process Control with Neural Networks"*) and the detection of wakes in radar images of the ocean surface (in *"Detection of Ocean Wakes in Synthetic Aperture Radar Images with Neural Networks"*, which was developed under an IR&D project). New theoretical and practical developments include the development of a successive approximation technique for handling sparse data sets and including known information into a network (in *"Multi-parameter Process Control with Neural Networks"*) and the application of the Logicon Projection Network™ (developed by us under an IR&D project) to the problem of character recognition in order to demonstrate the speed-up in training time over traditional networks such as back-propagation (in *"The Projection Neural Network"*).

Our research has led to a better understanding of the strengths and weaknesses of neural networks (NN's). We have shown, through empirical and analytic studies, the feasibility of scaling up to large problems. And we have developed networks capable of solving such large problems. We have shown the ability of NN' s to generalize well even when trained on a small set of data in comparison with the number of free parameters available to learn. And we have demonstrated this in a practical application to process prediction. Furthermore, we have compared the popular backpropagation NN to several other neural networks (radial basis functions, Reduced Coulomb Energy, Adaptive Resonance Theory, Logicon Projection Network™, Lynch-Granger Model), fuzzy logic and statistical techniques.

Over the past several years many researchers, including ourselves, have demonstrated the utility of neural network algorithms for the solution of problems for

which training on real data is essential. NN approaches form one extreme of a spectrum of algorithms. At the other extreme are physical models in which analytic formulae or equations are available to adequately describe the system to be solved. Fuzzy logic and expert systems form an alternate approach in which the physical knowledge is presented in the form of rules. An important direction for future research is the integration of all three approaches. This should lead to techniques which can incorporate the physical knowledge with training on experimental data to incorporate effects not modeled or to learn values of unknown parameters.

Another area prime for future research is the development of NN's capable of solving useful dynamic problems. The current algorithms for solving such problems are susceptible to getting stuck in local minima far from the desired solution. Careful design of the network structure tuned for each particular problem is likely to lead to efficient solutions.

In the field of intelligent image recognition careful tailoring of the network to the problem is likely to have an effective outcome. For image preprocessing, feature extraction and classification NN's offer the possibilities for construction of parallel algorithms which integrate key features necessary for the effective application of NN's to automatic target recognition (ATR) as well as other image recognition tasks: 1) insensitivity to local distortions as well as global (translation, rotation) transformations of images, 2) a design which facilitates detection of images based on a collection of features, only a portion of which may be present in any given image, and 3) the ability to train the network to learn the best features necessary for image recognition. Future research in this area is likely to yield large benefits in performance for image recognition systems.

Overview of Neural Network Theory

1. Classification of Neural Networks

Many neural networks can be classified into two basic classes: 1) those that divide the input space into open regions (usually hyperplanes); 2) those that divide the space with closed regions (hyperspheres, city block volumes, etc.). The first type includes standard perceptron-like nets which are usually feedforward nets trained by backpropagation. These suffer from slow training times because of the inability to initialize the weights close to the desired solution. On the other hand they have the advantage of being trained by error minimization, so that the network outputs will, given sufficient training data, approximate the Baye's conditional probabilities. The second type of network includes hypersphere classifiers, such as RCE, ART, and learning vector quantization. These networks do not ensure error minimization. On the other hand they learn quickly because each hidden node can be initialized to represent a prototype which encloses a decision region in the input space. This ensures that the network is near a good solution. Logicon's Projection Network™ combines both types of networks into one. It can initialize open or closed prototypes and still train by error minimization.

2. How does one choose which network to use?

Supervised or Unsupervised: If the data set is classified choose a supervised network or perhaps first use an unsupervised network to cluster the data and reduce the dimensionality and then use a supervised network.

For a supervised network, one must choose between feedforward or feedback nets with many variations in training style, counterpropagation net, cascade correlation net, neocognitron, Reduced Coulomb Energy (RCE) net, Adaptive Resonance nets (ART, ARTMAP), Boltzmann machine, Bidirectional Associate Memory (BAM), Brain State in a Box net, Hopfield net. Radial Basis Function nets such as the Probabilistic Neural Net (PNN), Learning Vector Quantization (LVQ), the Logicon Projection Network. This is not a complete list but does include most of the popular networks. They can be functionally grouped together in the following manner (note that there is some overlap of function though only the dominant mode of use is listed for the most part):

Optimization problems such as traveling salesman:
- Boltzmann machine

Autoassociation, pattern completion:
- Brain State in a Box
- Hopfield net
- Bidirectional Associative Memory
- Boltzmann machine

Feedforward nets, classification problems, function fitting, control

- Feedforward nets, usually hyperplane decision boundaries (error minimization)
  - Backpropagation & variants (slow training)
  - Cascade correlation (faster training)
  - Logicon Projection Network™ (faster training)

- Prototype formation, (no error minimization, tend to be fast training)
  - Learning vector quantization
  - RCE
  - ART
  - Counterpropagation
  - Logicon Projection net

Recurrent Backpropagation, dynamic problems, capturing contextual information, robotics, control

A general rule is to use the simplest network which will solve a problem. This accounts for the widespread use of backpropagation training. For most problems it can be adjusted to give a reasonable solution. When training speed is a major concern but error minimization is not, one may choose one of the prototype formation nets. The projection net forms prototypes and also minimizes the error.

An example

As an example, suppose one wishes to design an experiment to take a set of data from various sensors and classify some objects. Perhaps the objective is to discriminate a truck form a car based on seismic measurements. Let us say there are 100 sensor

measurements so that the dimension of the input space for our classifier is 100. The first question that arises is: how much data should be collected ? In order to estimate this, one must know how complex the problem is, something that in general cannot be found except by attempting to solve the classification problem. In practice, one uses past experience in solving similar problems as a guide for estimating the amount of data required. Ideally, one would use an iterative approach in which an initial data set is used to design and test a classifier and then the utility of increasing the size of the data set is studied.

Suppose we have decided to collect 200 data points, each point representing 100 inputs and one output probability (0 for car, 1 for truck). We must next decide the neural network to use. If we are interested in a fast but not necessarily optimized solution we might use a hyperspherical classifier, perhaps ART[1] or RCE. Such classifiers work well when the data falls into separable clusters for each in the input space. On the other hand, if we have reason to believe that the problem is not easily solved with clusters of prototypes or if we want to ensure error minimization to optimize the solution, we may use some modification of backpropagation training of a feedforward net. Assuming the latter, we must decide on the number of hidden layers and nodes in each layer.

There are no good heuristics for choosing the number of hidden layers. One can rely on Cybenko's result that one hidden layer is sufficient to map any bounded continuous function. But for some problems more layers may lead to shorter training times or the avoidance of local minima. For simplicity, let us assume one hidden layer. The number of nodes in this layer must be sufficient to match the complexity of the problem. The best procedure is to train and test the generalization performance (on a different data set) as one varies the number of hidden nodes. There are indications that more hidden nodes leads to shorter training times (measured in terms of number of training epochs)[2]. If one is willing to accept a generalization error of 0.1, then one knows that n_w/n_d should be roughly 0.1, if the network is trained for an infinite time. For this error, with a single hidden layer net of n_h hidden nodes, the condition $0.5\ n_h \approx 0.1$ cannot be satisfied. If it could (i.e. if we had at least 5 times the available data) we would use this to set a rough bound on the number of hidden nodes. We might decide to use this rule to guide us in choosing more data. On the other hand, the typical situation is that this is all of the available data. Even though we cannot satisfy the rule of thumb, the data set may offer useful information. We do know from this analysis, however, that the neural net should not be trained to completion. Careful attention must be paid to the generalization performance as training proceeds and the network should be stopped when this performance is optimum. Alternately, another regularization scheme such as weight decay can be used.

3. Rules of Thumb

Neural networks learn by example. Hence, to train a network (or any other classifier or function interpolator) one must determine the number of training data points

---

[1] Note that ART is not exactly a hyperspherical classifier but the gist of the algorithm can be considered just that.

[2] S. Fahlman, "An Empirical Study of Learning Speed in Back-Propagation Networks", CMU-CS-88-162, September 1988.

needed to learn a problem to a given level of accuracy. While there are few general results, there are some results which suggest the following rule of thumb:

The generalization error will be some simple function of the number of parameters , n_w, in the network (number of weights plus number of biases) divided by the number of training points, n_d.

$$e \approx n_w/n_d.$$

For a given number of data points and some desired error, a good rule of thumb is then to choose the number of parameters in the network according to the above formula. In practice, one can often get by with many more parameters than this. This arises because the number of effective parameters is actually smaller than n_w. By stopping training before completion (as for instance determined by the performance on a test validation data set) or by adding a weight decay term to the error function one introduces an implicit or explicit regularization[3]. Thus a neural network can begin with a larger number of parameters than that suggested by the rules of thumb described below and still avoid overfitting. By stopping the training process at a time at which the validation error is at a minimum, one may obtain a good generalization.

On the other hand, in order to estimate how many data points are needed one needs to understand the complexity of the problem as measured by the number of weights required to learn the training data. This is why neural nets (or any other method) require a lot of judgement in order to solve a problem. One must understand the complexity of the problem in order to choose enough data points and to estimate the number of parameters needed in the net. On the other hand, one cannot in general determine the complexity except by attempting to train the neural network and then judging its performance.

4. How Do NN's Fit with Statistical Estimation Theory?

The results of statistical estimation theory are generally valid for arbitrary parametric functional forms. Neural networks provide an example parametrization of a general function. The advantages of neural nets are that 1) they offer the capability of fitting any bounded continuous function all in a simply parametrized form and 2) they provide efficient and flexible training algorithms to learn the parameters by example.

Statistical methods usually begin by predicting the probability, p(x|c), that describes the distribution of data within class c. Bayes' rule is then used to obtain the probability of interest, p(c|x), the probability, given a data point x, that it should be classified as class c:

$$p(c|x) = p(x|c)\, p(c)/[\Sigma_c\, p(x|c)\, p(c)].$$

Neural nets, on the other hand usually operate by directly predicting the classification probabilities p(c|x). They are therefore more direct.

---

[3] L. Ljung, J. Sjoberg, "A System Identification Perspective on Neural Nets", 1992.

5. NN's can learn to approximate the Bayes' conditional probabilities:

The proof of this statement hinges on two conditions: 1) that the neural network be trained by error minimization and 2) that enough training data is available to sample the probability distributions.

Consider a two-class classification problem. Let the network be trained to output a desired value of 1 for class 1 and 0 for class 0. Then the error which is to be minimized is

$$E = \Sigma_{(c=1)} (1-p)^2 + \Sigma_{(c=0)} (0-p)^2 ,$$

where p is the neural net output and the sum over all data has been partitioned into the two classes. If a statistically significant sampling of data is available, the sums can be approximated by integrals over the probability distributions:

$$E \approx \int dx \; [\, p(1 \mid x)\, p(x)\, (1-p)^2 + p(0 \mid x)\, p(x)\, p^2 \,].$$

This can be rearranged to give

$$E \approx \int dx \; p(x) [\, p(1 \mid x) \; (1-p)^2 + p(0 \mid x) \; p^2 \,].$$

The point x is either class 0 or it is class 1, i.e. $p(1|x) + p(0|x) = 1$. Therefore,

$$E \approx \int dx \; p(x) [\, (p(1 \mid x) - p)^2 + p(1 \mid x) \; (1 - p(1 \mid x)) \,].$$

The second term is independent of the neural network and is a minimum error determined by the overlap of classes 0 and 1. The first term contains the only reference to the neural network. Hence the process of adjusting the neural network parameters to minimize the error is equivalent to adjusting the output of the network, p, to approximate the Bayes' conditional probability $p(1 \mid x)$.

6. What is the role of additional hidden layers?

In a standard back-propagation network, the neurons in the first hidden layer form hyperplanar boundaries in the original input space. If the network only has one hidden level, then the network output is a superposition of sigmoidal surfaces. The hyperplanar boundaries must be carefully placed such that their constructive and destructive interference will yield the correct result. One advantage of an additional hidden layer of neurons is that this interference is more easily avoided. As an example, two nodes in the first hidden level can combine to give a Gaussian-like response to one input. This can be repeated, using two nodes for each input dimension. A single second layer neuron can then sum up all of these nodes to form a Gaussian-like response in N dimensions. Any bounded and continuous function can be built up of a combination of such bumps. These bumps, or radial basis functions, are localized and hence can be designed not to interfere appreciably with neighboring bumps. The final output neuron sums up the radial basis function-like responses of the second hidden layer neurons.

Additional hidden levels may provide additional layers of abstraction, but there are few examples in the literature to illustrate this. Training times tend to increase as one adds hidden layers. A better alternative to the addition of layers to a network is a more intelligent choice of network structure. For example, a neural network can be constructed as a feature detector to preprocess the data. Additional layers can then be added to classify this processed data

7. Why Use Different Error Functions?

One of the main problems which slows down BP training is the saturation of the sigmoid where the derivative of the sigmoid is close to zero:

$$d\sigma/dw = \sigma (1-\sigma)$$

approaches 0 as σ approaches 0 or 1. The problem is that the derivative of the error with respect to the weight is proportional to the derivative of the sigmoid:

$$dE/dw = d/dw \{0.5 \Sigma [p - p']^2 \} = \Sigma [p-p'] dp/dw,$$

where p is the neural network output and p' is the desired value. The sum is over all output nodes and training data. dp/dw always contains derivatives of sigmoids.

One solution to this difficulty is to replace the error function, which is normally chosen to be quadratic in the difference of the NN output from the desired output, by some other function whose derivative does not vanish as rapidly. An example is the entropic error measure:

$$E = \Sigma [p \log p + (1-p) \log 1/(1-p)].$$

This error vanishes when the neural network output p equals the desired output p', as it should. The derivative with respect to the upper level weights now contains terms of the form

$$dp/dw /[p (1-p)] .$$

The derivative of the sigmoid which comes out of the numerator now cancels the denominator. Thus, the final layer of weights do not get bogged down from saturation of the output level nodes.

8. Regularization

Regularization is the process of adding smoothness constraints to an approximation problem[4]. An example of such a constraint is the addition of a term to the error which is to be minimized proportional to an integral of the squared curvature

[4] T. Poggio and F. Girosi, "A Theory of Networks for Approximation and Learning", A. I. Memo No. 1140, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1989.

(second derivative) of the function to be fit. For one-dimensional functions, minimizing an error function containing this term and the sum squared deviations of the data from the predicted function gives rise to a cubic spline fit. The constant of proportionality for the regularization term controls the compromise between the degree of smoothness of the solution and the closesness to the data.

An alternate regularization term for a neural network is a weight constraint which might prevent the weights from growing too large (large weights can give rise to steep sigmoids and hence allow the fitting of detailed structure). This can be accomplished with the addition of a term to the error which is quadratic in the weights. This is often referred to as weight decay, since upon gradient descent, an additional term -gw appears in the training rule which tends to decay the weights toward 0.

Also, neural networks have a built-in, implicit, regularization procedure: simply stop training before completion. Normally one stops at a point which minimizes the generalization error (the error obtained on a test data set independent from that on which the network has been trained).

9. Why does backpropagation plateau out?

Backpropagation can plateau out or get stuck in bad local minima for several reasons. If the error surface is very shallow, the gradients are small and the descent to a minimum will be slow. This region of very slow progress shows up as a plateau in the error vs training time curve. If one watches the motion of the decision surfaces (hyperplanes) as one trains, one typically sees one hyperplane being brought into play at a time. Rapid training occurs after a hyperplane has gotten close to the right position and amplitude. Training then slows down significantly as a new hyperplane, which is typically far away from where it should be, begins creeping in.

Plateaus often occur when the sigmoids are saturated. In other words, the hyperplane is so far away from where it should be that the input to a node is very large in magnitude. The derivative of the sigmoid is then very close to one and so the gradient of the error with respect to the weights and hence the weight change will be small.

One does not in general have a way of knowing whether or not one is in a local minimum or a plateau. Assuming that one has checked that the sigmoids are not saturated, one could still have a flat error surface or bad local minima because of the complexity of the problem.

10. Scaling Issues

The neural network structure has been shown to reduce the number of computations required for calculating the gradient of the error. A separate issue is the number of iterations needed for a gradient descent calculation to reach an error minimum. There are few theoretical results along these lines. because the number of training iterations required N_t is very dependent upon the nature of the problem to be solved. For some problems, such as the separation of two multidimensional Gaussian distributions,

the number of iterations scales linearly with input dimension. Furthermore, N_t tends to decrease as the number of weights is increased for a fixed input dimension (i.e. the number of hidden nodes is increased) because there is a greater probability that the hyperplanes will be near desired positions. Other problems, such as the less physical N-dimensional parity problem are known to require an exponentially increasing number of iterations to train as the input dimension is increased. On the other hand, these methods apply to the simple backpropagation training algorithm. More sophisticated modifications of the algorithm have been developed which do not suffer from long training times for high-dimensional problems.

11. Theoretical Results

The following four results are the broadest in scope.

1. NN outputs are estimates of Bayes a posteriori probabilities when trained to minimize squared error or cross entropy [5]. Example: If an output node is trained to equal 1 for class 1 inputs and to equal 0 for class 0 inputs then after training on a statistically significant sample of data the values of the node will in general lie between 0 and 1 and will approximate the probability p(c | x) that input x should be classified as class c.

2. Cybenko's Theorem[6,7] : A feedforward network with sigmoidal nonlinearities and one hidden layer of nodes can approximate arbitrarily well any continuous bounded function. It can also approximate some discontinuous functions, provided that the discontinuity is relatively benign, e.g. a step function can be approximated but a fractal function can not.

3. A fully recurrent network is shown to be able to approximate arbitrarily well a dynamical system[8]. That is a network with n+m nodes is capable of reproducing a differentiable function of time **y**(t) (where **y** is an n dimensional vector) over a bounded time interval. m is a number greater than or equal to the number of time steps for which one wants to resolve the function.

4. An additive fuzzy system can approximate any continuous function on a compact domain to any degree of accuracy[9]. The fuzzy system approximates the function by covering it with fuzzy patches where each fuzzy rule defines a patch.

The remaining results are rather specific in the assumptions and do not necessarily hold true for backpropagation trained nets in general.

---

[5] M. Richard, R. Lippmann, "Neural Network Classifiers Estimate Bayesian a posteriori Probabilities", Neural Computation, v. 4, 1991, pgs. 461-483.

[6] G. Cybenko, "Approximations by Superpositions of a Sigmoidal Function", *Mathematics of Control, Signals, and Systems*, v. 2, 1989, pgs. 303-314,.

[7] K. Hornik, M. Stinchcombe and H. White, 'Multilayer feedforward networks are universal approximators", Neural Networks, 1989, pgs. 2-5, 359-366.

[8] L. K. Li, "Approximation Theory and Recurrent Networks", IJCNN 92 II, 1992, pgs. 266-271.

[9] B. Kosko, "Fuzzy Function Approximation", IJCNN 1992 I, 1992, pgs. 209-213

This following result is based on the Vapnik-Chervonenkis (VC) dimension, a measure of the capacity of a network. The VC dimension of a network is the largest number of points that can be shattered by the network, i.e. the number of possible ways of dividing the input points into two classes. For linear thresholding neurons, the VC dimension is shown to be at least as large as the number of weights in the network for a large number of hidden nodes.

5. Under certain broad conditions, the generalization error e of a learning system with VC dimension D trained on n_D random examples of an arbitrary function will be no worse than a bound of the order of D/n_D (with high confidence)[10]. While this gives a bound, it does not indicate what the average generalization error will be, and it may be too pessimistic in many cases.

6. In a network with n_W independent parameters (weights and thresholds), the average predictive entropy (-log(1-error)) can be shown to converge to 0 as n_W/n_D for large n_D where n_D is the number of training data[11]. The result holds generally for any learning machine, not just neural nets. When the error is small:

$$\text{error} \approx n_W/n_D.$$

The proof of this depends on the important and probably invalid assumption that the probability of reaching any weight configuration is equally likely. This is fine if one's training algorithm simply jumps randomly from one set of weights to another until it reaches a solution. But it is probably not true with backpropagation gradient descent training.

The generalization error made by any machine independently of its architecture (including neural networks) is related to the number of training points as a power law:

$$\text{error} \sim n_D^{-p}$$

If there is one and only one possible setting of the adjustable coefficients of the machine, i.e. the weights of the neural network which give the "correct" solution for every possible input, and if the training data is free of noise, i.e. it always knows the correct answer, then p=1 (case A). Then the proportionality constant is equal to the number of adjustable coefficients n_W, so that

$$\text{error} = n_W/n_D,$$

which is the Vapnik-Chervonenkis (VC) upper bound.

If the set of possible combinations of nn weights which correctly solves the problem has a volume of nonzero measure in weight space, and again, the training data is free of noise, then p=2 (case B). If there is one and only one possible set of correct

---

[10] E. Baum, D. Haussler, "What Size Net Gives Valid Generalization?", Neural Computation, v. 1, 1989, pgs. 151-160.

[11] S. Amari, "Universal Property of Learning Curves under Entropy Loss", IJCNN 1992 II, 1992, pgs. 368-373.

weights as in A but the training data is noisy so that sometimes the network is trained with an incorrect answer, then p=1/2 (case C). Finally, if the correct answer is itself noisy or stochastic so that the desired output for each input point is determined probabilistically, then p=1 (case D), but the error decays asymptotically to a nonzero value.

In case A, since the dependence of the proportionality constant on the number of weights is known, the expression for the error can be inverted to determine the bounds on the number of required hidden nodes or training data points to achieve a tolerable error. For example, if the tolerable error is e, then we need a number of weights equal to $n_W = en_D$.

This expression can also be derived through another approach: Each hidden layer node is connected to n input nodes and therefore draws a hyperplane in n dimensions which can pass through any n distinct points. In other words, each neuron can exactly fit n data points. But, if every point is fit exactly in training, the generalization will be poor. Therefore, the number of weights should be less than the number of training points so that $n_W < n_D$.

7. For a linear net with no hidden units, analytic results can be derived for the generalization error as a function of training time[12]. If the initial weights are sufficiently small the generalization error has a unique minimum, corresponding to an optimal training time. More generally there are at most n minima for a net with n inputs and n outputs.

8. For a net with no hidden units, n inputs and one output, and whose inputs and weights take on only the values +1 and -1, perfect generalization occurs with some probability if the number of data points is greater than αn. (There are several claims that α ranges from 1.24 to 2.0)[13]. The probability goes to 0 as n approaches infinity. Perfect generalization means that given a set of n_D data points generated from a known neural network, another neural network trained on this data will converge to the exact same weights as the original network.

While this result is not very useful as it stands it does suggest the possibility that more general neural networks may also show the same qualitative behaviour; i.e. the required number of data points for good generalization grows only linearly with the input dimension.

9. The probability of reaching a given weight vector starting from some initial weight vector as a function of training time can be derived as a Fokker-Planck (diffusion) equation. In addition, the probability of escaping from a local minimum as a function of

---

12 P. Baldi, Y. Chauvin, "Temporal Evolution of Generalization during Learning in Linear Networks", Neural Computation, v. 3, 1991, pgs. 589-603.

13 E. Baum, Y. Lyuu, "The Transition to Perfect Generalization in Perceptrons", Neural Computation, v. 3, 1991, pgs. 386-401.

training time can be expressed as a Fokker-Planck equation. The results are verified with experiments on the XOR problem.[14]

10. The error in a feedforward neural network with one hidden layer of sigmoidal nodes can be shown to a have a bound which depends on the number of hidden nodes n: The squared error is of the order of c/n when the network is approximating functions which satisfy a particular smoothness condition. c is a constant which depends on the function to be fit and may depend on the input dimension d; but there are many functions for which c can be shown to have a less than exponential dependence on d. In contrast, if such a function is fit to a series expansion with n fixed terms, the squared error can not be made smaller than order $(1/n)^{2/d}$. Thus, for series expansions, as the input dimension is increased, the number of parameters must grow exponentially with d in order to maintain a given error. The neural network, on the other hand, can potentially achieve the same error without an exponentially increasing number of hidden nodes. The key difference is that for a neural network, the basis functions are adaptable, whereas for series expansion methods, the basis functions are fixed[15].

This key difference between the ANN and a series expansion of fixed basis functions can be explained as follows: For the ANN one can specify a given number of hidden nodes, n. The neural network then provides an expansion with n sigmoidal functions. But the functions themselves are not fixed from the start; they are varied through training of the weights $\mathbf{w}_i$' (shown below). On the other hand, with the series expansion approach, one chooses n definite basis functions. The basis functions are not free to change, therefore, in general, a greater number of them are needed to fit the same function.

Series expansion: $f(\mathbf{x}) = \Sigma_i \, w_i \, f_i(\mathbf{x})$.

ANN: $f(\mathbf{x}) = \Sigma_i \, w_i \, \sigma_i(\mathbf{w}_i' \cdot \mathbf{x} - v_i)$.

The proof is detailed but the gist of it can be illustrated as follows. Suppose that the function we wish to approximate can be represented by a one-hidden layer ANN with a very large number of hidden nodes, let us say a million. Now the question is how well can we approximate this with only n nodes, where n is less than a million:

$$f_n(x) = 1/n \, \Sigma_i \, g_i \, , \quad \text{where} \quad g_i = n \, w_i \, \sigma_i \, .$$

Let the n functions g_i be sampled randomly from the million g's that make up the desired function. Then if we take an ensemble average of such samplings we will reproduce the function f(x): $\langle f_n(x) \rangle = f(x)$. Furthermore the average value of the squared error can be expressed as

$$\langle (f_n(x) - f(x))^2 \rangle = \langle (1/n \, \Sigma_i \, g_i)^2 \rangle - f(x)^2$$

---

[14] T. K. Leen, G. B. Orr, "Weight-Space Probability Densities and Convergence Times for Stochastic Learning", IJCNN 1992 IV, 1992, pgs. 158-164

[15] A. Barron, "Universal Approximation Bounds for Superpositions of a Sigmoidal Function", to appear in IEEE Transactions on Information Theory.

$$= [\langle g_1 g_2 \rangle + 1/n \langle g_1^2 - g_1 g_2 \rangle] - [\langle g_1 g_2 \rangle + 1/N \langle g_1^2 - g_1 g_2 \rangle]$$

$$\approx 1/n \langle g_1^2 - g_1 g_2 \rangle$$

$$\approx 1/n \langle (g_i - f(x))^2 \rangle$$

$$\leq c/n.$$

The term $\langle g_1 g_2 \rangle$ is an ensemble average over components with differing indices and is equal to $f(x)^2$ in the limit of large N. The last line above assumes a constant c which bounds both g and f. Now, given that the ensemble average of the squared error is less than or equal to c/n it must be true that there is at least one selection of g's which has an error less than or equal to c/n.

The above proof shows that there is a selection of weights such that a neural network with n hidden nodes will produce a squared error less than c/n. These weights are typically found by gradient descent training which adjusts the weights to minimize the squared error. On the other hand, if one attempts to fit the function with n terms of an expansion with fixed basis functions, then one will in general not have the best n terms. Unless the basis functions are allowed to vary during training, the limit of c/n on the error is not guaranteed. Indeed for problems of higher input dimension d, the probability that the first n terms in an expansion are the correct terms must be exponentially small (something like e^d) since the number of possible basis functions grows exponentially with d. To be guaranteed a good error, the number of fixed basis functions must grow exponentially with the dimensions of the problem. This is the curse of dimensionality which is overcome by using variable basis functions.

11. Pineda's Argument on NN Computational Efficiency

The neural network structure (weighted sums of nonlinear functions of weighted sums) provides a framework for mapping arbitrary nonlinear functions of a set of inputs to a set of outputs. In addition, this structure allows calculation of the gradient of the output error with respect to the free parameters in a form which reduces the computational requirements by a factor of N, where N is the number of free parameters (weights plus thresholds) in the network. Because the most common optimization techniques require the calculation of this gradient, neural networks provide a factor of N reduction in the number of computations required to solve optimization problems. This may be the most important reason that backpropagation (error minimization) algorithms have made such an impact in neural computing.

The calculation of an arbitrary function of N parameters in general requires at least on the order of N calculations (such as additions or multiplications). The gradient of an error function formed from such a function in general then requires of the order of N^2 calculations, since there are N components to the gradient and each component requires on the order of N calculations. As an example, consider a polynomial of order N-1, which has N free parameters $w_0 .. w_{N-1}$:

$$y(x) = w_0 + w_1 x + w_2 x^2 + ... + w_{N-1} x^{N-1} .$$

For problems in which y(x) is to be fit to some desired value y^a over a fixed training set of points x^a, a conventional approach is to minimize the quadratic error over the training set:

$$\varepsilon = \frac{1}{2}\sum_a \left(y(\mathbf{x}^a) - y^a\right)^2.$$

The gradient of the error is then

$$\frac{d\varepsilon}{dw_n} = \sum_a \left(y(\mathbf{x}^a) - y^a\right) n\, w_n \left(x^a\right)^{n-1},$$

which requires of the order of N calculations (to evaluate $y(x^a)$) for each training point.

On the other hand, a neural network with one hidden layer of nodes has an output of the form

$$y(\mathbf{x}) = \sigma\left[\sum_{j=1}^{n2} w'_j\, \sigma\left(\sum_{i=1}^{n1} w_{ij}\, x_i\right)\right].$$

The number of free parameters is N = n1 n2 + n2 = (n1+1)n2. We will consider the typical situation in which the number of input nodes n1 is much greater than one, then N ≈ n1 n2. Because of the iterated structure of the network, the evaluation of the gradient of the error simplifies to a product of two pieces for each data point:

$$\frac{d\varepsilon}{dw'_j} = \sum_a \delta^a\, x'^a{}_j \qquad \frac{d\varepsilon}{dw_{ij}} = \sum_a \delta_j^a\, x_i^a\,,$$

where

$$\delta^a = (y(\mathbf{x}^a) - y^a)\,\sigma' \qquad \delta_j^a = \delta^a\, w'_j\, \sigma'_j.$$

The argument of σ is understood to be the weighted sum of inputs coming into the output node and the argument of σ_j is understood to be the weighted sum of inputs coming into the hidden node labeled by j. Calculation of the gradient is basically reduced to calculation of δ^a which requires of the order of N computations. δ_j^a then requires another N computations for a total of the order of 2N (which is still of the order of N).

Scaling of Back-Propagation Training Time to Large Dimensions

(published in IJCNN'90 Proceedings)

Abstract

The training time for the back-propagation neural network algorithm is studied as a function of input dimension for the problem of discriminating between two overlapping multidimensional Gaussian distributions. This problem is simple enough (it is linearly separable for distributions of equal standard deviation which are not centered at the same point) to allow an analytic determination of the expected performance, and yet it is realistic in the sense that many real-world problems have distributions of discriminants which are approximately Gaussian. The simulations are carried out for input dimensions ranging from 1 to 1000 and show that, for large enough N, the training time scales linearly with input dimension, N, when a constant error criterion is used to determine when to terminate training. The slope of this linear dependence is a function of the error criterion and the ratio of the standard deviation to separation of the two Gaussian distributions. The closer the separation, the longer is the required training time. For each input dimension a full statistical treatment was implemented by training the network 400 times with a different random initialization of weights and biases each time. These results provide insight into the ultimate limitations of a straightfoward implementation of back-propagation.

1. Introduction

The popularity of the back-propagation (BP) neural network training algorithm can be traced in part to the simplicity and ease of implementation of this paradigm which is capable of forming nonlinear and non-overlapping classification regions (Cybenko, 1988, Lippmann, 1987). Its utility lies in its practical application to large classification and optimization problems where the reduced calculational complexity of BP over non-neural methods can make a difficult problem tractable (Pineda, 1989). While several empirical studies have been carried out to investigate the scaling of problems which are relatively difficult to learn with BP, such as the parity function (Fogaca, Kramer, 1988) or the simpler linearly separable majority function (Ahmad, Tesauro, 1988), the former has limited applicability to realistic problems and the latter has been restricted to problems of relatively small dimension. The present study is intended to extend this work to explore a slightly more realistic problem in the realm of input dimensions ranging from 1 to 1000 dimensions.

2. The Input: Multi-Dimensional Gaussians

The problem considered is that of two overlapping Gaussian distributions in an input space of dimension N. For simplicity, only spherically symmetric distributions are considered in this report. Two such distributions are linearly separable. That is, one can draw a hyperplane (a line in two dimensions, a plane in three, etc.) in the input space such that all points on one side of the hyperplane are to be classified as class 1 and all points on the other side are to be classified as class 2. Because the two distributions

overlap, there will be missclassifications. But this partition of the input space (Bayes' criterion) minimizes the errors. The fraction of missclassified points can be calculated as follows: The two distributions are described by Gaussians of the form:

$$p_1(\mathbf{r}) = 1/(\sqrt{(2\pi\sigma_1^2)^N}\ \exp[-(\mathbf{r}-\mathbf{r}_1)^2/(2\sigma_1^2)], \quad (1)$$

$$p_2(\mathbf{r}) = 1/(\sqrt{(2\pi\sigma_2^2)^N}\ \exp[-(\mathbf{r}-\mathbf{r}_2)^2/(2\sigma_2^2)], \quad (2)$$

where $p_1(\mathbf{r})$ ($p_2(\mathbf{r})$) gives the probability of finding a point, known to be of class 1 (2), at position $\mathbf{r}$ in the N dimensional space. $\mathbf{r}_1$ and $\mathbf{r}_2$ are the centers of the two distributions, and σ_1 and σ_2 are the respective standard deviations. The probability of interest is the probability, $p(1 \mid \mathbf{r})$, that a given point $\mathbf{r}$ should be classified as class 1. By Bayes' relationship, this can be expressed in terms of the probabilities given in equations 1 and 2. For example,

$$p(1 \mid \mathbf{r}) = p_1(\mathbf{r})\, p_1 / [p_1(\mathbf{r})\ p_1 + p_2(\mathbf{r})\ p_2], \quad (3)$$

where p_1 and p_2 are the 'a priori' probabilities of finding class 1 or 2 irrespective of position and are therefore proportional to the number of points of each class. Each point is assumed to be either of class 1 or of class 2; the 'a priori' probabilities add up to unity. The probability that a point at $\mathbf{r}$ should be classified as type 1 can be expressed as

$$p(1 \mid \mathbf{r}) = 1/[1 + p_2/p_1\ p_2(\mathbf{r})/p_1(\mathbf{r})]. \quad (4)$$

According to Bayes' rule (Duda, Hart, 1973), for the situation in which there is an equal penalty for missclassifying classes 1 and 2, the point at $\mathbf{r}$ should be classified as class 1 whenever $p(1 \mid \mathbf{r}) > p(2 \mid \mathbf{r})$. This is seen to be equivalent to the statement

$$p_1(\mathbf{r})\ p_1 > p_2(\mathbf{r})\ p_2. \quad (5)$$

For simplicity we will consider the situation in which there are equal numbers of points in each distribution. Furthermore, the standard deviations of the two distributions will be assumed equal:

$$p_1 = p_2,$$
$$\sigma_1 = \sigma_2 = \sigma. \quad (6)$$

Then the classification boundary is, according to equations 1, 2 and 5, described by an equation linear in $\mathbf{r}$:

$$\mathbf{r} \cdot (\mathbf{r}_1 - \mathbf{r}_2) = 1/2\,(\mathbf{r}_1 + \mathbf{r}_2) \cdot (\mathbf{r}_1 - \mathbf{r}_2), \quad (7)$$

which describes a hyperplane in N dimensions midway between the centers of the two distributions and normal to the vector that joins the two centers. In the more general case for which the standard deviations of the two distributions are different, the classification

boundary would be a quadratic surface. For the current situation the fraction of points missclassified, can be expressed as an integral of $p_1 p_1(\mathbf{r})$ over all points to the right of the hyperplane (assuming class 1 is on the left and class 2 is on the right) added to the corresponding integral of $p_2 p_2(\mathbf{r})$ over all points to the left of the hyperplane. The result for f, the fraction of points correctly classified, is given in terms of the complementary error function:

$$f = 1 - 1/2 \, \mathrm{erfc}(\delta r/(\sigma\sqrt{8})), \qquad (8)$$

where δr is the distance between the two centers.

3. Implementation

Because the problem considered is linearly separable, it can in principle be solved by a two-layer neural network (no hidden layer) (Lippmann, 1987). Indeed, it is easy to see, from equations 1, 2 and 4 just how to choose the weights so that the network 's output value is equal to the desired classification probability, $p(1 \mid \mathbf{r})$. However, for the present study, a network with one hidden layer has been used so that comparisons may be made among networks having different numbers of neurons in order to understand the scaling of training time with both problem size, N, and with the size of the network.

Accordingly, the network consists of an input layer having N nodes, a middle layer having N' nodes (which is varied from 1 to 20), and a single output node which represents the probability that the input point should be classified as class 1. The value of the i^{th} input node represents the i^{th} coordinate value of the input point which was restricted to lie in the range from -1 to +1. The output value was trained to be 1 or 0 according to whether the input point belonged to class 1 or 2. A training set of unlimited size was used. That is, each point in the distribution was randomly selected from the entire space for each trial. The network was tested on a different set of 500 randomly selected points.

A standard backpropagation algorithm adopted from Rumelhart and McClelland (1986) was used to train the net. A learning rate of 0.3, a bias learning rate of 0.3 and a momentum coefficient of 0.7 were used. For each experiment, the network was trained 400 times, with a different selection of randomly chosen initial weights (between -1 and +1). This allows us to describe the statistical distribution of training times.

4. Results

The distributions of number of trials to learn is shown in Figure 1 for the case in which $\delta r/\sigma = 4$ ($\delta r = 0.4$, $\sigma = 0.1$). This is for a network with one node in the middle layer which is trained until 90% of the points in the test set are correctly classified (longer training could produce a fraction f = 98% according to equation 8). Several points can be observed from this. As the input dimension, N, is increased the distributions shift to longer training times, as expected. The shift is apparently linear with N. Moreover, the

spread of the distribution increases (and the peak correspondingly decreases) as N increases. This is shown more clearly in Figure 2, where the mode, mean and standard deviations of the distributions are shown. As can be seen, for input dimensions ranging from 100 to 1000, the required number of trials is approximately linear in N. The distributions are slightly skewed as reflected in the fact that the mean is larger than the mode. Also shown in Figure 2 is the increase in slope when the network is trained to yield fewer missclassifications.

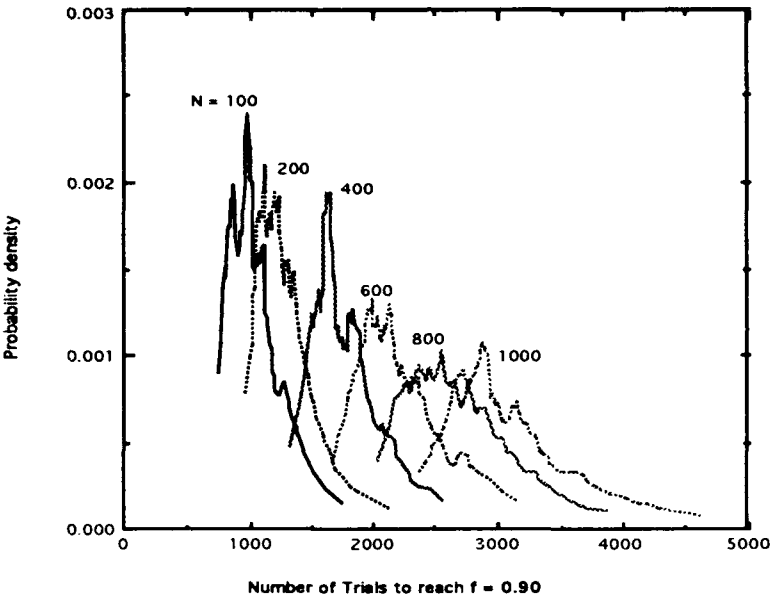


Figure 1. Distribution of number of training iterations for various input dimensions. $\delta r/\sigma = 0.4/0.1$, f=90%.

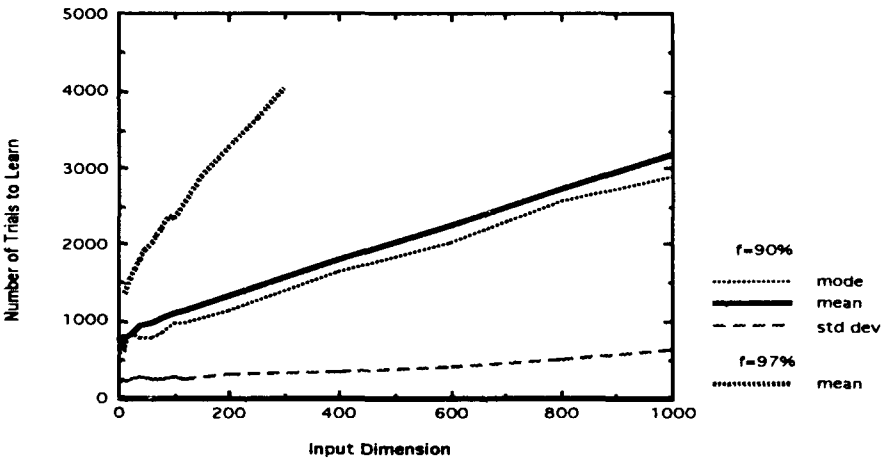


Figure 2. Scaling of number of trials to learn to a level of 90% and 97% correct for two overlapping Gaussian distributions. Each point represents a statistical sampling of 400 training sessions. $\delta r = 0.4$, $\sigma = 0.1$.

The slope of the line which approximates the large dimension behavior is dependent upon the ratio of the spacing to standard deviation of the Gaussians, as is shown in Figure 3. The smaller is the ratio, the more closely spaced are the distributions, for fixed σ, and the scaling with input dimension is more steep. The more the overlap of the two distributions, the more difficult is the problem for the back-propagation algorithm. Note that the slope is obtained from the distributions of training times for N=100 and N=200 only. The linear dependence at large N has been checked for $\delta r/\sigma = 4$ and 2 only. For the experiments which generated the data in this figure the criterion for ending the training session was that the network classify correctly the test data set as best as is possible (the Bayes' limit of equation 8) to within 1%. This is a more stringent requirement which results in longer training times and steeper slopes for the curve of training iterations vs N. The behavior depicted in the figure is roughly an inverse dependence of the slope on $\delta r/\sigma$, i. e. slope $\propto \sigma/\delta r$. For the example of N = 10, shown in Figure 4, the training time was also shown to scale as $\sigma/\delta r$.

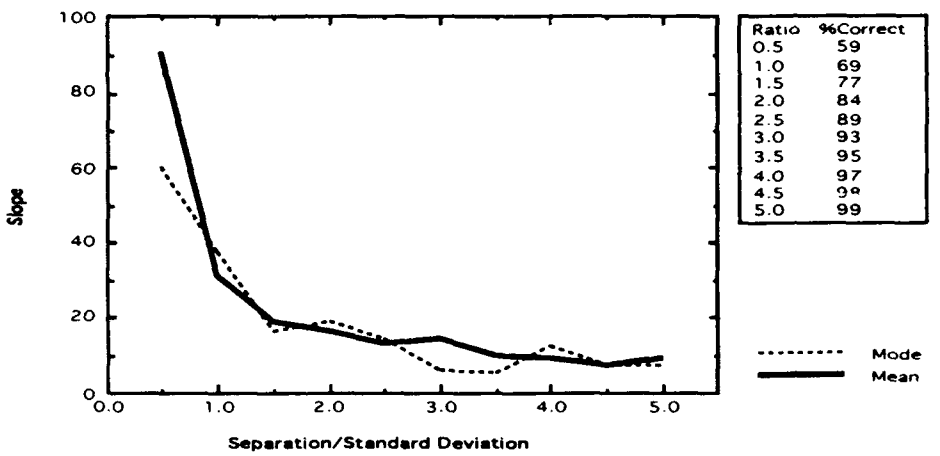


Figure 3. Dependence of the slope of the curve for training iteration vs N on the ratio $\delta r/\sigma$. The networks are trained to within 1% of the fraction correct given by Equation 8. The table in the upper right corner indicates the values of f (chosen to be within 1% of Bayes' limit) for each value of $\delta r/\sigma$ used.

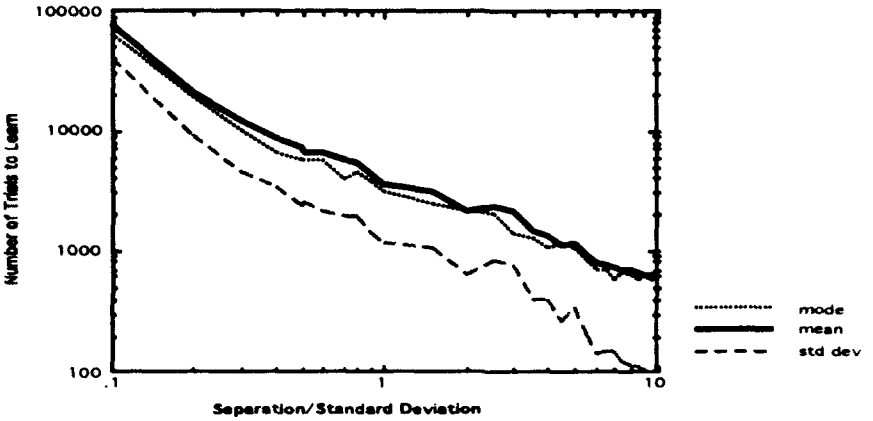


Figure 4. Dependence of number of training iterations on separation of Gaussians for 10 input dimensions. The networks are trained to within 1% of the fraction correct given by Equation 8.

The dependence on the number of nodes in the middle layer of the network is shown in Figure 5. There is an initial decrease in number of trials required followed by a gradual increase. Presumably, even though one node is sufficient, the number of ways of obtaining a solution is greater as one increase the number of nodes. This is countered by the larger weight space which must be searched. Presumably the curve reflects the competition between these two effects.

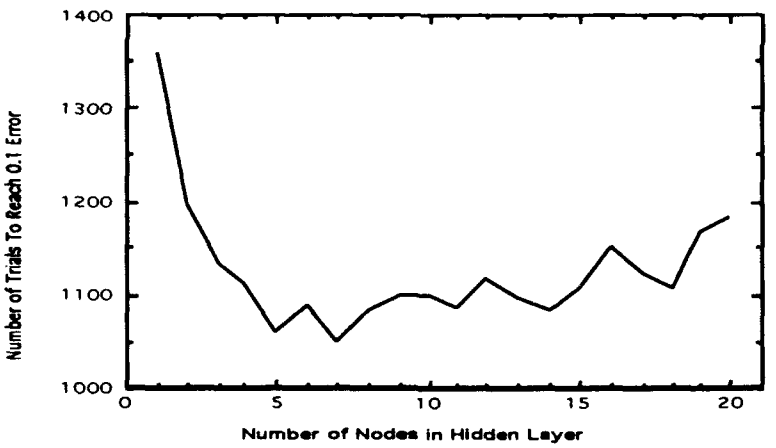


Figure 5. Dependence of mean number of trials to reach a sum squared output error of 0.1 on the number of nodes in the hidden layer. N = 10, $\delta r/\sigma = 0.4/0.1 = 4$.

5. Conclusions

Calculation of the times to train a feedforward network using the back-propagation algorithm have been performed for the simple problem of two overlapping Gaussian distributions. Results have shown a linear scaling with input dimension. The slope of the line is roughly proportional to the ratio of the standard deviation to the separation of the two distributions. As the ratio increases, the overlap between the distributions grows, and more iterations are required to train the network. The fact that the slope increases as the separation decreases or the error criterion becomes more stringent (rather than simply giving a translation of the curve to larger numbers of trials) is not understood, but it indicates that the relative difficulty of solving problems of different dimensions is dependent upon the error criterion and the characteristics of the problem (parametrized in this example by $\delta r/\sigma$).

Furthermore, as the input dimension increases, the training time distribution function changes. This function gives the probability per unit range of the number of training trials that a given set of initial weights, chosen at random, will lead to a network that trains to a fixed error level. This distribution becomes more spread out as N increases; the standard deviation increases. In particular, the tail of the distribution grows larger, making it more likely to require a larger number of trials to learn. Any empirical studies of the training time must train the network a correspondingly larger number of times (with different initial weights) for large dimensional problems in order to make any statistically meaningful statements.

While these empirical conclusions are useful for understanding the training time required for a simple class of problems, the theoretical challenge remains to explain the results analytically. Does the behavior continue for larger input dimensions? How is the scaling modified for problems that are not linearly separable or that have multiply disconnected regions in the input space? Can one derive the scaling behavior as a function of a more general measure of problem complexity? These questions remain to be anwered.

6. References

Ahmad, S., G. Tesauro, "Scaling and Generalization in Neural Networks: A Case Study", Proceedings of the 1988 Connectionist Models Summer School, 1988, Touretzky, Hinton, Sejnowski, Morgan Kaufmann Publishers, San Mateo, California.

Cybenko, G., Continuous Valued Neural Networks with Two Hidden Layers are Sufficient, 1988, Tufts U. preprint.

Duda, R. O., P. E. Hart, Pattern Classification and Scene Analysis, John Wiley and Sons, 1973, New York.

Fogaca, M., A. Kramer, B. Moore, "Scalability Issues in Neural Networks", Neural Networks, 1988, pg.1, Suppl 1.

Lippmann, R., 1987, "An Introduction to Computing with Neural Nets", *IEEE ASSP Magazine*, 1987, v.4(2), pgs. 4-22.

Pineda, F. J., "Recurrent Backpropagation and the Dynamical Approach to Adaptive Neural Computation", Neural Computation, 1989, v. 1, pgs. 161-172.

Rumelhart, D. E., J. L. McClelland and the PDP Research Group, Parallel Distributed Processing, MIT Press, 1986, Cambridge, Massachussetts.

7. Acknowledgements

This research was sponsored by Geo-Centers, ONR and DARPA under contracts GC-1710-89-R-001 and N00014-89C-0257.

Classification and Function Fitting as a Function of Scale and Complexity

1. Introduction

Understanding neural network performance as a function of the scale and complexity of the problem it is trained to solve is important in extrapolating the capability of neural networks in solving complex large scale problems from a knowledge of their performance on simpler, smaller scale problems. To measure this performance, we refer to two general applications of neural networks, namely classification and fitting: Classification involves the division of a set of vectors into two or more sets or "classes", and fitting is the approximation of a vector function of a vector argument.

The performance of the neural network on these two problems is measured by its ability to generalize: If the neural network is trained on a set of inputs (the training set), how well does it perform on another set of inputs (the test set)? This performance measure is a function of the number of training inputs and the number of intermediate layer neurons in the network. As the number of points in the training set is increased, the classifier becomes more accurate in its predictions on the testing set. Similarly, a larger classifier (more nodes in a neural network, higher order polynomial, etc.), is able to discern greater detail or successfully classify a more complicated problem. The claim that too many nodes in the intermediate layer of a neural network can result in overlearning is also investigated. By varying the number of training points and the classifier size, we can study their effects on generalization. Finally, the network performance on the fitting problem is compared to the performance of a polynomial least squares fitting algorithm with a similar learning scheme.

The neural network used for both the classification and the fitting problem is a standard, three layer back-propagation feedforward network with one input node, a variable number of intermediate level nodes, and one output node.

2. Classification of 2 Regions in 1-D

2.1 Approach

Perhaps the simplest example of a classification problem is the division of a 1-D (scalar) input into two classes. Let the single input x vary from 0 to 1 and let the output class y_d ,the desired output, be

$y_d = 0$ if $x < 0.5$ and
$y_d = 1$ if $x > 0.5$.

Learning the appropriate class for each x is done by example: The training set is composed of n pairs (x,y) chosen at random, where the input x ranges from 0 to 1, and y_d is the correct class of the input x and is set to 0 or 1. The neural network has a single output y designed to reproduce the correct output y_d. Its error is defined as

$$E = (y - y_d)^2 / 2 .$$

The network randomly chooses a pair (x,y) from the n pairs, and then learns by changing its weigths and thresholds through the gradient descent method, which is designed to minimize the error by moving opposite the direction of the gradient of the error. The performance, defined as the generalization capability, is measured by the fraction of test points misclassified. Iterating through the entire test set of 1000 points, also chosen randomly, for each point,

if $y - y_d < 0.5$, then the test point is considered correctly classified, and

if $y - y_d > 0.5$, then the test point is considered misclassified.

The fraction misclassified is the total number of misclassified test points in both classes divided by the total number of test points.

Since both the training set and the initial neural network weights are chosen randomly, a single training session on one training set will not provide a statistically significant measure of performance. It is necessary to choose a number of different training sets and initial random weights for each n, and to average the fraction of points misclassified over these different sets of training points and weights.

2.2 Results

The performance of the neural network is shown in Figure 1 where the fraction of test points misclassified by the network is plotted against the number of input points used to train the network. As discussed above, each point in Figure 1 represents an average over 20 different sets of weights for each of 20 different sets of n training points. With only one training point (n=1), the network fails to generalize. Since it only sees a point from one class, it missclassifies all the points from the other class, a 50% misclassification. With two points, there is still a large probability (0.5) that both points will be from the same class, hence the error is still large. As n increases, the probability that all points will be from the same class becomes small and the points will tend to span the entire range of inputs values, thus reducing the fraction misclassified.

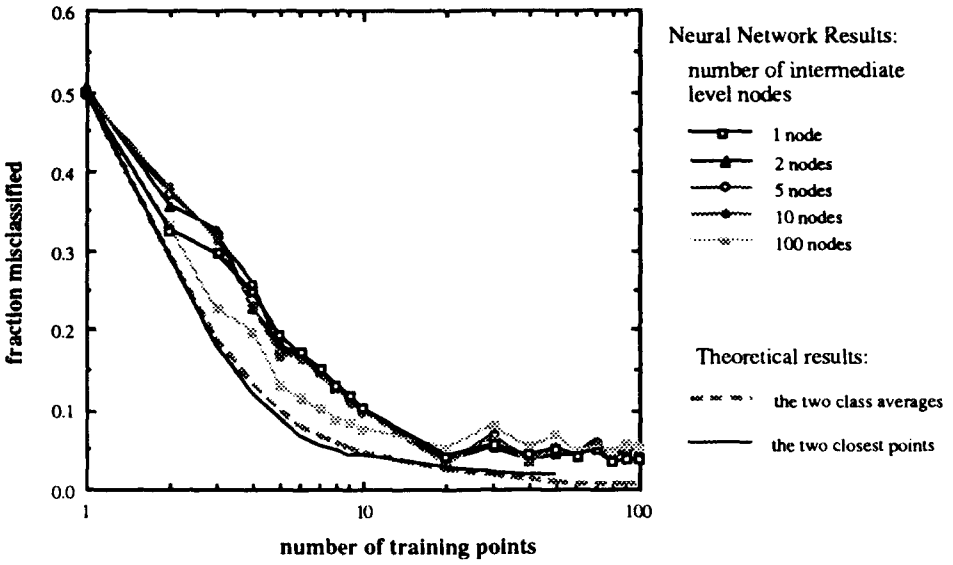


Fig. 1: Classification of Two Regions by a Neural Network-Performance as a function of the number of training points

The dashed line in Figure 1 is an exact theoretical value of the minimum possible fraction of test points misclassified when a classifier is trained on n points, and averaged over all possible choices of n training points. This is calculated by finding the average position of the training points in each class and drawing the boundary between the two classes halfway between the two average positions. The fraction misclassified corresponds to the distance from this halfway mark to the correct boundary between the two classes (0.5). This result is derived in the appendix and is obtained by averaging the fraction misclassified over all the possible choices of input points. The solid line is another theoretical calculation of the minimum possible fraction misclassified, also averaged over all possible choices of training points. In this calculation, first, the two closest points of opposite classes are found, and then the boundary between the two classes is drawn halfway between these two points. This result is also derived in the appendix and is nearly equal to the other result (see Figure 1). For large n, the second theoretical calculation of the least possible classification error is inversely proportional to the number of training points n, i.e.,

$$f_n \sim 1/2n ,$$

for large n. In other words, for this simple classification problem, in order to reduce the fractionmisclassified by a factor of two, one needs to use twice as many training points. We do not claim that this result is general; its utility and applicability to other classification problems needs to be investigated. However, the results from this particular

problem suggests that probably, for some classes of problems, the fraction of points misclassified will be inversely proportional to the number of training points.

The neural network performance is consistent with this result; its error is generally slightly above this theoretical limit. Since we ceased training after reaching an error of 0.01, we did not gauge the best possible performance of the neural network. After a longer training session, the network error should move even closer to the theoretical limit.

Increasing the number of intermediate layer nodes m and therefore also the number of weights and thresholds has little effect on the performance. Since only one intermediate layer node is required to separate two regions, increasing m has little effect on performance. This can be seen in Figure 1 but also more clearly in Figure 2 where the fraction misclassified is plotted against the number of intermediate layer nodes. For any n, the fraction of points misclassified is essentially constant for all m.

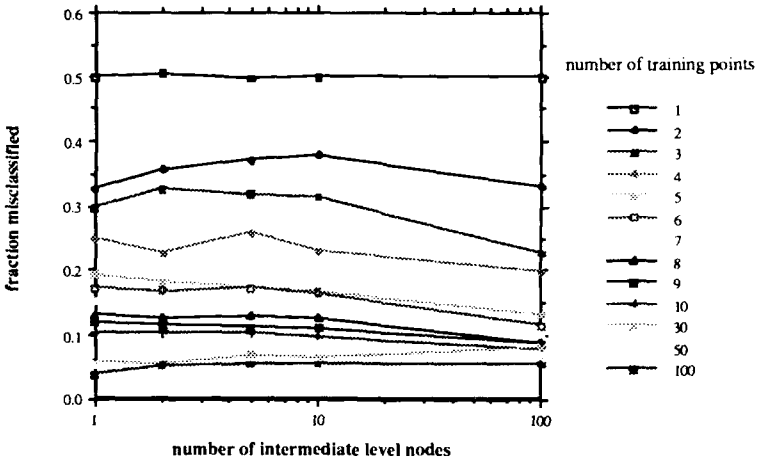


Fig. 2: Classification of Two Regions by a Neural Network-Performance as a function of the number of intermediate level nodes in the network.

"Overlearning" is not observed even for very large m, (m = 100). Note that this result holds for any number of training points (n = 1, 2, 3, 4, ..., 10, 30, 50, 100). Our explanation is that in learning to classify, a neural network translates and orients "hyperplanes" until they coincide with the boundaries which divide spatial regions corresponding to different classes. Since we start from randomly selected weights and thresholds, i.e., random positions and orientation of hyperplanes, it is unlikely that many hyperplanes will appear in the vicinity of the boundary. Instead, usually a single hyperplane will be brought to the boundary during the learning process, and once it is properly located and oriented, other hyperplanes will be subdued, i.e. the weights and thresholds to other intermediate level nodes will be decreased. Hence, overlearning does

not occur in this problem, since it would require the presence of more than one hyperplane near the boundary between the two regions.

3. Fitting: Estimating a Function

3.1 Approach

The simplest example of a fitting problem is the fitting of a scalar function f of a scalar input x. f can be any simple smooth, bounded function, bu' our choice of a sinusoid will allow us to vary the complexity of f(x) in future studies by varying the wavelength of the sinusoid. Let

$$f(x) = A + B\sin(ppx), \quad p = 0,1,2...$$

Our objective is to study the generalization performance of the neural network and compare it to a polynomial least squares fitting algorithm.

The neural network and the polynomial least squares fit are trained on a set of n training points with the gradient descent method. Each training point x is chosen randomly in the interval from 0 to 1, and its desired output is the value of the function at x, f(x). The gradient descentmethod applied to the polynomial fitting scheme is the same as that applied to the neural network.

The error made by the neural network for each training or test point is:

$$E = [\, y(x) - f(x) \,]^2 \,/\, 2 \ ,$$

where y(x) is the output of the neural network for the input point x.

For a polynomial of order m, and for each point, the error made by the polynomial in estimating the f(x) is analogous:

$$E = [\, p_m(x) - f(x) \,]^2 \,/\, 2 \ ,$$

where $p_m(x)$ is the polynomial approximation (fit) of f(x) for the input point x:

$$p_m(x) = a_0 + a_1 x + a_2 x_2 + ... + a_m x_m \ .$$

In learning, the coefficients in the polynomial are treated in the same way as the weights in the neural network. At each iteration of the learning process, the coefficients are changed in a direction opposite to that of the gradient, i.e.,

$$Daj = -g\, dE / daj \ ,$$

where g is a gain parameter. By iterating through the training set, we minimize the error.

For this study, we consider the simplest case, i.e. p = 0 and A = 0.5, where f(x) becomes a constant and the fitting problem reduces to the problem of approximating a constant function.

The performance for the fitting problem is measured through the error (rather than the fraction of misclassified points) for both the polynomial fitting algorithm and the neural network, by averaging over the error of the entire test set of 1000 points.

As in the case of the classification problem, a single random selection of training points and weights will not provide a statistically accurate measurement of performance. Therefore, we will average the performance over 20 different selections of weights for each of 20 different selections of n training points.

3.2 Results

In Figure 3, the generalization error of the network is plotted against the number of training input points. Since two points are necessary to determine a line, one training point is insufficient for training either a network or a polynomial to approximate a line. Two training points are the minimum number needed, but since the points are chosen randomly, they may be near each other or span a limited range of values. Therefore the addition of more training points results in some enhancement of the performance, but the enhancement saturates as n is increased especially when n becomes large enough so that the input set spans the range of x values.

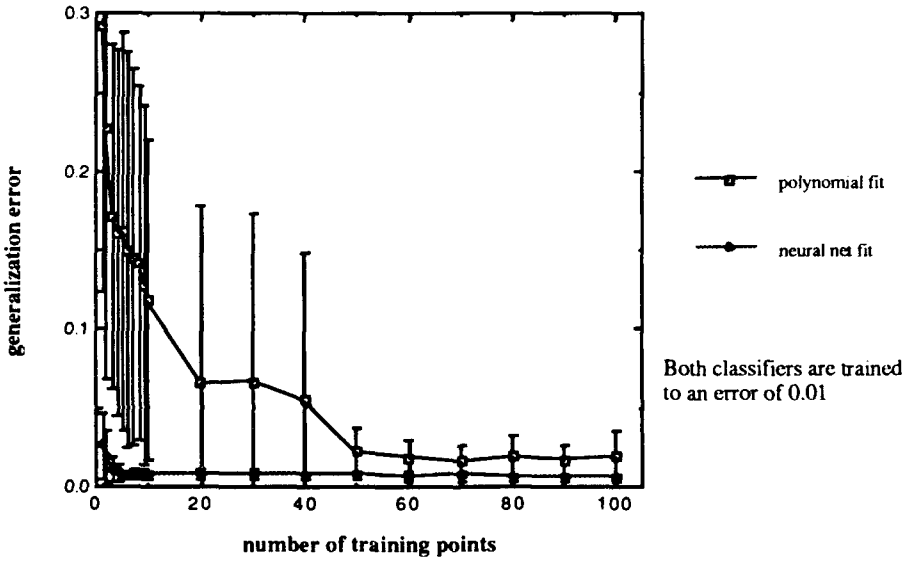


Fig. 3: Function fitting- Comparison of the neural network fit to the polynomial fit with gradient descent learning.

The performance of the neural network is compared to that of the polynomial least squares fit in Figure 3. The error made by the polynomial is always greater than the error made by theneural network especially when few input points are used in training. With fewer than 10 input training points, the polynomial error is an order of magnitude greater than the neural network error. Actually, this result may also be true for larger numbers of training points, but since we had to stop training after the error reached or went below a value of 0.01, the limits of the capability of the neural network were not tested. In fact, after n = 10, the neural network error is already below 0.01.

Again, the addition of intermediate layer nodes has little effect and overlearning is absent even with as many as 100 intermediate nodes. Our explanation for this is the same as the one we gave for the absence of overlearning in the classification problem. As can be seen in Figure 4, the generalization error is insensitive to changes in m for $m \geq 4$. For $m < 4$, the number of training points is insufficient for generalization even though the network may learn the training set perfectly.

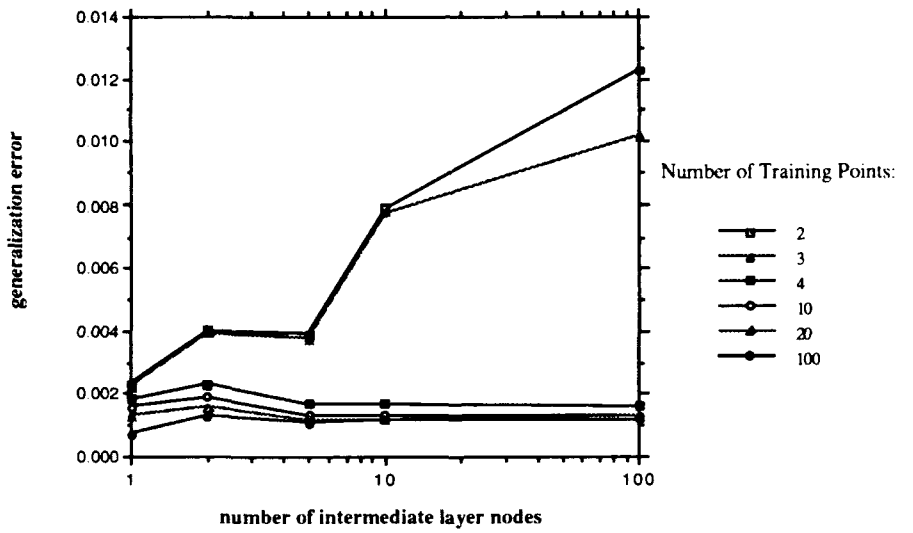


Fig. 4: Function (line) fitting with a neural network - Generalization error vs. number of hidden layer weights.

In Figure 5, the number of trials to learn the training set to an error of 0.01 or less is plotted against the number of training points for the neural network and the polynomial fitting algorithm. Note that when training on very few training points (one or two), the performance on the test set may be poor even though the time to learn the training set is short, i.e. generalization is difficult. The time it takes the neural network to learn to the 0.01 error level is an order of magnitude below that of the polynomial fitting algorithm for most n and particularly for large n. Also, for n larger than 10, increasing n does not

effect the learning time of the neural network, but it does increase the learning time of the polynomial fitting algorithm at least up to n=90.

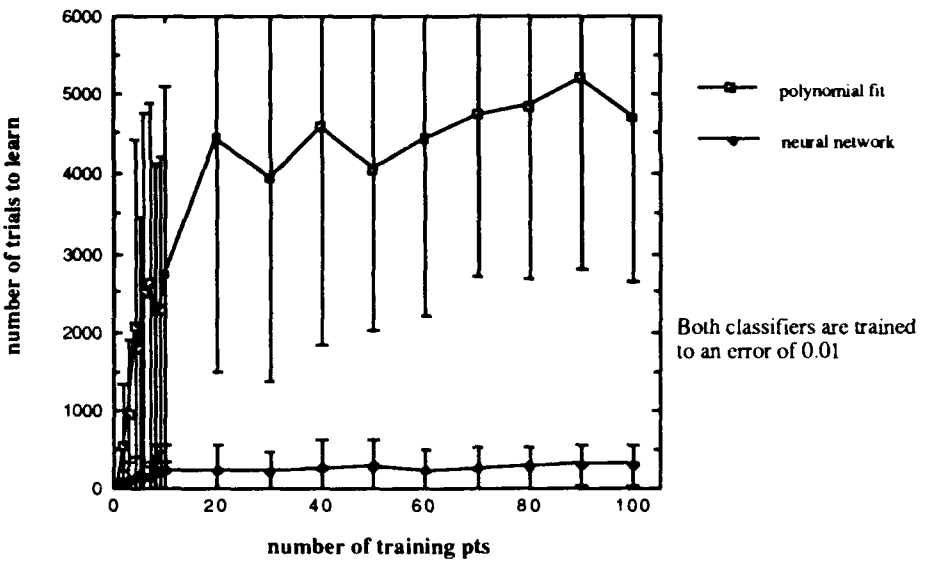


Fig. 5: Function fitting: Comparison of training times of a neural network with a polynomial fit using gradient descent learning.

This shows the strength of the neural network in fitting a function with relatively few training points and with an error well below the error made by the polynomial least squares algorithm.

4. Conclusions

We have demonstrated the strength of the neural network in the classification and fitting problems. With relatively few training points, the neural network finds a classification which generalizes successfully. In addition, for the simple problems in this study, we have shown that the neural network generalization capability is insensitive to the number of intermediate layer nodes in the network, as long as the minimum number of required nodes is exceeded. Overlearning is absent and does not present a danger when training for long times or with many intermediate layer nodes. Much of this may be problem dependent, but it suggests that for certain classes of problems, the neural network can exhibit this strong generalization capability with very few inputs, along with the absence of danger of overlearning. We have also shown the superiority of the network's performance over a conventional polynomial least squares algorithm for the fitting problem.

5. Appendix

Analytic calculation of the boundary between two classes in one dimension

The boundary between two classes of points in a one dimensional problem can be calculated analytically in a number of ways. In this appendix, two different solutions are shown.

The first approach draws the boundary halfway between the two closest points of opposite class. For convenience, let the points from class 1 and class 2 be denoted by x and x' respectively, and chosen randomly in the ranges

$$0 \le x \le 1 , \text{ and } 0 \le x' \le 1 .$$

Then, the boundary is drawn halfway between the smallest x and x':

$$x_b = (x - x') / 2 .$$

Since the actual boundary is at zero, the fraction of points misclassified will be the ratio of the distance between zero and x_b and the total range of x and x' which is equal to 2:

$$E(x,x') = \frac{1}{2}|x_b| = \frac{1}{4}|x - x'| .$$

The probability that q-1 points will be chosen between x and 1 and that one point will be between x and x+dx is:

$$p_q(x)\,dx = q\,(1 - x)^{q-1}\,dx .$$

dx is the probability that a point will be chosen between x and x+dx, and $q\,(1 - x)^{q-1}$ is the probability that the other q-1 points out of q points will fall between x and 1. Similarly,

$$p_r(x')\,dx' = r\,(1 - x')^{r-1}\,dx' .$$

is the probability that r-1 points will be chosen between x' and 1 and that one point will be between x' and x'+dx'. Note that

$$\int_0^1 p_q(x)\,dx = \int_0^1 p_r(x')\,dx' = 1 .$$

The average error made in choosing q points in class 1 and r points in class 2 is found by integrating over all possible choices of x and x':

$$e(q,r) = \int_0^1 dx \int_0^1 dx' p_q(x) p_r(x') E(x,x')$$

$$= \frac{qr}{4} \int_0^1 dx \int_0^1 dx' (1-x)^{q-1} (1-x')^{r-1} |x-x'|$$

We can exchange x and x' with 1-x and 1-x'

$$e(q,r) = \frac{qr}{4} \int_0^1 dx \int_0^1 dx' \; x^{q-1} x'^{r-1} |x-x'|$$

and eliminate the absolute value sign:

$$e(q,r) = \frac{qr}{4} \int_0^1 dx \left\{ \int_0^x dx' \; x^{q-1} x'^{r-1} (x-x') + \int_x^1 dx' \; x^{q-1} x'^{r-1} (x'-x) \right\}$$

After integration, we get

$$e(q,r) = \frac{1}{4(q+r+1)} \left[\frac{q}{r+1} + \frac{r}{q+1} \right]$$

To find the average error for n points chosen from either class, we must consider all possible choices of q and r which add to n. Since

$q + r = n$, let

$r = n - q$.

The number of ways in which q points will be chosen in one class and n-q points will be chosen from the other class is

$$\frac{n!}{q!\,(n-q)!}$$

each of which has a $1/2^n$ probability of occurance. The error made for each choice of n and q is e(q,n-q) for $0 < q < n$. If $q = 0$ or $q = n$, then all the points chosen fall into one of the two classes. This will occur with a probability of 2^{1-n} . In these cases, all the points belonging to the other class will be misclassified, so that the error will be 1/2 . Therefore, the contribution to the total error from the $q = 0$ and $q = n$ cases will be 2^{-n}. The total error or the fraction of n points misclassified is

$$e_n = \frac{1}{2^n} + \frac{1}{2^n} \sum_{q=1}^{n-1} \frac{n!}{q!\,(n-q)!} e(q,n-q)$$

$$e_n = \frac{1}{2^n} + \frac{1}{4(n+1)2^n} \sum_{q=1}^{n-1} \frac{n!}{q!\,(n-q)!} \left[\frac{q}{n-q+1} + \frac{n-q}{q+1} \right]$$

which is the sum of the errors made when q=0, q=n, and 0<q<n. The two sums can be rearranged to give:

$$e_n = \frac{1}{2^n} + \frac{1}{4(n+1)2^n} \left[\sum_{q=0}^{n-2} \frac{n!}{q!\,(n-q)!} + \sum_{q=2}^{n} \frac{n!}{q!\,(n-q)!} \right]$$

which can be readily summed:

$$e_n = \frac{1}{2^n} + \frac{1}{4(n+1)2^n} \left[2^{n+1} - 2n - 2 \right]$$

Therefore, the fraction of points misclassified out of n randomly chosen points is:

$$e_n = \frac{1}{2(n+1)} + \frac{1}{2^{n+1}}$$

For large n, the fraction misclassified falls off as 1/2n .

The second approach draws the boundary halfway between the average positions of the two classes of input points. Consider input points x from class 1 and x' from class 2 in the ranges:

$$-\frac{1}{2} \le x < 0 \quad , \quad \text{and} \quad 0 < x' \le \frac{1}{2} \quad .$$

Let p points be chosen randomly from class 1 and q points from class 2, and let the average positions of the two classes of points be

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \quad \text{and} \quad \bar{x'} = \frac{1}{n'} \sum_{j=1}^{n'} x'_j \quad .$$

The boundary between the classes is drawn at

$$x_b = \frac{1}{2} \left[\bar{x} + \bar{x'} \right] ,$$

and the fraction of points misclassified for a particular choice of x_b is the fraction of points which fall between x_b and the true boundary at 0:

$$f(x_b) = |x_b| = \frac{1}{2}|\bar{x} + \bar{x}'| .$$

This error occurs for all values of x_i and x_j for which the boundary ends up at x_b. The average fraction missed is found by integrating over all possible values of x_i and x_j and x_b:

$$\bar{f} = 2^{n+m} \int_{-1/2}^{1/2} dx_b \int_{-1/2}^{0} dx_1 \int_{-1/2}^{0} dx_2 \ \ldots \int_{1/2}^{0} dx_n \int_{0}^{1/2} dx'_1 \int_{0}^{1/2} dx'_2 \ \ldots \int_{0}^{1/2} dx'_m \ f(x_b)\delta(x_b - \tfrac{\bar{x} + \bar{x}'}{2})$$

where 2^{n+m} is the normalization constant. The integral over x_b eliminates the delta function and we get

$$\bar{f} = < \frac{1}{2}|\bar{x} + \bar{x}'| > ,$$

where $< >$ is a shorthand for 2^{n+m} times the n+m integrals over all x_i and x_j. The integrand can be expressed in terms of the step function Θ

$$|\bar{x} + \bar{x}'| = (\bar{x} + \bar{x}')\Theta(\bar{x} + \bar{x}') - (\bar{x} + \bar{x}')\Theta(-\bar{x} - \bar{x}').$$

The step function, in turn, can be expressed as

$$\Theta(y) = \int_{-\infty}^{\infty} \frac{dk}{2\pi i} \frac{e^{iky}}{k - i\varepsilon} \quad \text{or} \quad \int_{-\infty}^{\infty} \frac{dk}{2\pi i} \frac{e^{-iky}}{-k - i\varepsilon}$$

in the limit that ε approaches zero. Therefore, the fraction misclassified can be expressed as

$$\bar{f} = \int_{-\infty}^{\infty} \frac{dk}{2\pi i} \frac{1}{k - i\varepsilon} \frac{1}{2} < (\bar{x} + \bar{x}')e^{ik(\bar{x}+\bar{x}')/2} - (\bar{x} + \bar{x}')e^{-ik(\bar{x}+\bar{x}')/2} > .$$

Integrating by parts and combining, we get:

$$\bar{f} = -2 \int_{-\infty}^{\infty} \frac{dk}{2\pi} \frac{1}{(k - i\varepsilon)^2} < e^{ik(\bar{x}+\bar{x}')/2} > .$$

The integral represented by the braces is over all x_i and x_j' and is a product of n+m integrals of the form

$$2 \int_{-1/2}^{0} dx \, e^{ikx/2n} \quad \text{or} \quad 2 \int_{0}^{1/2} dx' \, e^{ikx'/2n'} .$$

Therefore, the resulting integral is the product:

$$< e^{ik(\bar{x}+\bar{x}')/2} > = \left[\frac{sin(k/8n)}{k/8n}\right]^n \left[\frac{sin(k/8n')}{k/8n'}\right]^{n'} ,$$

and the fraction misclassified is

$$\bar{f} = -2 \lim_{\varepsilon \to 0} \int_{-\infty}^{\infty} \frac{dk}{2\pi} \frac{1}{(k-i\varepsilon)^2} \left[\frac{sin(k/8n)}{k/8n}\right]^n \left[\frac{sin(k/8n')}{k/8n'}\right]^{n'} .$$

Now, if we perform a binomial expansion of the products

$$\left[\frac{sin(k/8n)}{k/8n}\right]^n \left[\frac{sin(k/8n')}{k/8n'}\right]^{n'} = \left(\frac{4n}{ik}\right)^n \left(\frac{4n'}{ik}\right)^{n'} \left(e^{ik/4n} - 1\right)^n \left(1 - e^{-ik/4n'}\right)^{n'}$$

$$= \left(\frac{4n}{k}\right)^n \left(\frac{4n'}{k}\right)^{n'} i^{n-n'} \sum_{m,m'} \binom{n}{m}\binom{n'}{m'} (-1)^{m+m'} e^{\frac{ik}{4}\left(\frac{m}{n} - \frac{m'}{n'}\right)}$$

we can integrate term by term and get

$$\bar{f} = \frac{n^n n'^{n'} (-1)^n}{2(n+n'+1)!} \sum_{m,m'} \binom{n}{m}\binom{n'}{m'} (-1)^{m+m'} \left(\frac{m}{n} - \frac{m'}{n'}\right)^{n+n'+1} ,$$

where the sum is restricted to m/n > m'/n' . Since n+n' = N,

$$\bar{f} = \frac{n^n (N-n)^{N-n}}{2(N+1)!} \sum_{m,m'} \binom{n}{m}\binom{N-n}{m'} (-1)^{n+m+m'} \left(\frac{m}{n} - \frac{m'}{N-n}\right)^{N+1} ,$$

where the sum is again restricted to m/n > m'/(N-n).

A Comparison of Classifiers: The Neural Network, Bayes-Gaussian, and k-Nearest Neighbor Classifiers

1. Introduction

The utility and performance of various classification schemes, both classical and neural network, are strongly dependent on the type of problem to which they are applied. Comparisons of the application of these classifiers to a particular problem yield results which are specific to that problem and are often inapplicable to other problems. Therefore, it is useful to investigate the broad classes or types of problems which are particularly easy or difficult to solve with each classifer.

We confine this work to three classifiers, namely the Bayes-Gaussian, the k-nearest neighbor and the neural network applied to the general static categorization (or classification) problem, namely the categorization of many objects into a few classes.

For the purpose of this comparison, we find it useful to represent the problem of the categorization of objects into classes geometrically: Given n properties of each object to be used for classification, each object can be considered as an n-dimensional input vector x plotted in an n-dimensional "discriminant" space where each dimension corresponds to a property of the object. If the vectors belonging to each class fall into the same regions of space, then the properties (discriminants) can be successful in distinguishing between the various classes. These regions can be of arbitrary shape and geometrical complexity, including multiply-connected or stringy regions. The problem of discrimination is equivalent to the problem of separating this space into regions corresponding to a single class, i.e., drawing boundaries which separate the vectors belonging to one class from vectors of all other classes. The performance of any classification scheme in a particular problem depends on how the classifier draws boundaries in this space and whether or not its scheme is suited to the geometry of the regions formed by each class of input vectors.

In practical applications, performance may not be the only criterion in choosing the appropriate classifier. Two other important considerations are time and storage, the time it takes to implement the classifier (training plus testing) and the amount of storage required by the classification algorithm after learning. For some problems, the time or storage may not be a factor, but in general such criteria must also be considered in weighing the merits of a classifier in solving a particular problem.

2. Description of the Bayes-Gaussian and k-Nearest Neighbor Classifiers

2.1. Bayes and Bayes-Gaussian Classifier

The Bayes classification method is based on a simple relationship from probability theory. In order to describe this relationship, we begin with some simple definitions: Let p(A) be defined as the probability of occurrence of event A, and p(A|B) be defined as the conditional probability of occurrence of A given that B has occurred. Let p(A,B) be defined as the probability of occurrence of both A and B.

To derive this relationship, we note that p(A,B), the probability of occurrence of both A and B is the probability of occurrence of B times the probability of occurrence of A given that B has occurred and vice versa, i.e.

$p(A,B) = p(A|B)\, p(B)$, and also

$p(A,B) = p(B|A)\, p(A)$.

By equating the two equations above, we get the simple relationship between the conditional probabilities p(B|A) and p(A|B) which is the basis of Bayes classification scheme:

$p(A|B) = p(B|A)\, p(A) / p(B)$.

Therefore, the probability that a given point x belongs to class 1 can be expressed in terms of the probability that a class 1 point will be found at the position x:

$p(1|x) = p(x|1)\, p(1) / p(x)$,

where p(1) is the fraction of total points which belong to class 1, and p(x) is the probability of picking the point x from the distribution.

For convenience, we choose 2 classes; the following expressions can be easily extended to more than 2 classes if needed. Then, the normalization condition that every point x must belong to some class, i.e. class 1 or class 2

$p(1|x) + p(2|x) = 1$

leads to the expression

$p(1|x) = p(x|1)\, p(1) / [\, p(1)\, p(x|1) + p(2)\, p(x|2)\,]$

and a similar equation for p(2|x). Therefore, given any known distribution of each class of inputs, p(x|1) and p(x|2), and the relative probabilities of occurrance of class 1 and class 2 points, p(1) and p(2), we can calculate the probability that any given point x belongs to class 1 (or class 2).

The Bayes-Gaussian classification scheme is based on the assumption that the distribution of each class of inputs, p(x|1) and p(x|2), can be approximated by a Gaussian distribution. In an n-dimensional space, where each dimension corresponds to a discriminant input, an n-dimensional Gaussian must be fit to the distribution of each class. The mean of the Gaussian distribution corresponding to each class is the average value of the inputs which belong to that class. Forexample, the mean position of the class 1 distribution is

$$\langle x_i \rangle = (1/N)\, S_a\, x_i(a)\,, \text{ or}$$

$$\langle x \rangle = (1/N)\, S_a\, x(a)$$

where x(a) is the ath class 1 input point in the training distribution and $\langle x \rangle$ is the average class 1 input point. Then, to determine the width of the Gaussian along each dimension, we must calculate the covariance matrix C of the class 1 inputs, which consists of all possible correlations of inputs summed over the class 1 distribution:

$$Cij = (1/N)\, S_a\, (x_i(a) - \langle x_i \rangle)\, (x_j(a) - \langle x_j \rangle)\,, \text{ or}$$

$$C = (1/N)\; S_a\, (x(a) - \langle x \rangle)\, (x(a) - \langle x \rangle)\,.$$

where C is expressed as a dyadic. The Gaussian distribution can be written in terms of the inverse of C, C^{-1}, the determinant of C, |C|, and the average class 1 input vector $\langle x \rangle$

$$p(x|1) = \exp [\, -(x_k - \langle x \rangle) \Sigma C^{-1} \Sigma (x_k - \langle x \rangle) / 2\,] / [(2p)^{n/2}\, |C|^{1/2}]$$

where C^{-1} is the inverse matrix in dyadic form. The distribution is expressed as a conditional probability; p(x|1) is the probability that a class 1 point will be found at the position x. This procedure is repeated for each class to find p(x|2), p(x|3), etc.. Then, by using the normalized Bayes relationship discussed above, we can find p(1|x), p(2|x), etc., i.e. the probability that a given test point x belongs to a particular class.

We are free to set the threshold or cutoff which determines the minimum p(1|x) at which a given point x will be catagorized as class 1. By varying the threshold we can vary the boundary between the class 1 and class 2 regions.

2.2. k-Nearest Neighbor Classifier

The k-nearest neighbor classifier determines the categories of any point in discriminant space according to the category of its k-nearest neighbors. Each input vector x in the training set is a vector in the n-dimensional discriminant space and belongs to some known class. For convenience, choose just 2 classes. We calculate the Pythagorean distance from each test point to every training point, i.e.

$$d_{ij} = |\, x_i - x_j \,|$$

where x_i and x_j are any two points in the training and test sets, respectively, and d_{ij} is the distance between them. Then, for each test point, we find the k smallest distances, i.e. we find the k-nearest training points. The classification of each point is determined by the categories of its k-nearest neighbors. A "majority rule" determines the category of a site according to the majority of thecategories of its nearest neighbors. But, in general, the category of a site can be based on any number of nearest neighbors m from 1 to k. For example, for k=5, a site can be categorized as class 1 if at least 1, 2, 3, 4, or all 5 nearest neighbors are class 1 (i.e. m = 1,2,3,4, or 5). In this case, the majority rule corresponds to m=3. Thus, the correct classification of one class of inputs can be enhanced at the expense of the misclassification of the other class.

3. Performance

3.1. Bayes-Gaussian

If the distribution of input vectors of each class in discriminant space is known a priori, then the Bayes classifier (not necessarily Gaussian) is known to provide the optimum classification. However, if the distribution of input vectors for each class is not known a priori, and some distribution (e.g. Gaussian) is assumed, then the Bayes classifier may fall short of this optimum, especially when the assumed distribution does not match the actual one. The success of the Bayes-Gaussian classifier depends largely on how closely the actual distribution of input vectors of each class resembles a Gaussian. Since the classification scheme allows the adjustment of the spread of the Gaussian along every dimension, any ellipsoidal region (cigar-shaped in 3-D) can be constructed to house a particular class of inputs.

Therefore, the Bayes-Gaussian classifier performs well if the discriminant space is composed of one simply-connected, convex region with simple boundaries corresponding to each class of inputs. Conversely, it performs poorly if it encounters multiply-connected regions corresponding to one class or regions with complicated boundaries, especially concave boundaries. In Figure 1, we show some simple 2-D example problems in discriminant space which the Bayes-Gaussian classifier will succeed or fail in classifying.

In Example 1, we show two convex overlapping regions which belong to different classes. The mean position of these regions can be the same or different and their shapes can vary from very thin cigar shapes to spherical shapes. The Bayes-Gaussian will be generally successful with this kind of problem, since it will choose the appropriate mean and spread for each region and determine a probability that any given point belongs to either class. However, its success in determining the classification of "difficult" points, such as those in the overlapping region depend on how closely the actual distribution can be approximated by a Gaussian. For example, homogeneous (uniform) distributions or skewed distributions can cause misclassifications of points in overlapping or near boundary regions.

In Example 2, we have introduced a simple cross-like region of black dots enclosed in a white dot region. The Bayes-Gaussian will fail in this problem for two

reasons: The cross is a "concave" region, and the spread in the distribution of black dots is not uniform in each direction. In other words, the mean position of the distribution of the black dots can easily be determined to be the center of the cross but the spread in the horizontal and vertical directions are ambiguous: Near the center the spread in either direction is large but away from the center the spread in one direction becomes tight. If the classifier picks a tight Gaussian for the black dots, then it will miss the ones at the points of the cross; if it picks a broad Gaussian, then it will misclassify white dots near the boundary as black.

In Example 3, the Bayes-Gaussian classifier will fail terribly since the mean for thedistribution of either black dots or white dots will fall inside the other region! Even though the distribution of both black and white dots is simple and well separated from each other, there is no accurate way of fitting one Gaussian to either distribution. The only way to improve the performance of this classifier in this problem is to know, a priori, that there are two regions of black and white dots and to know which dot belongs to which region.

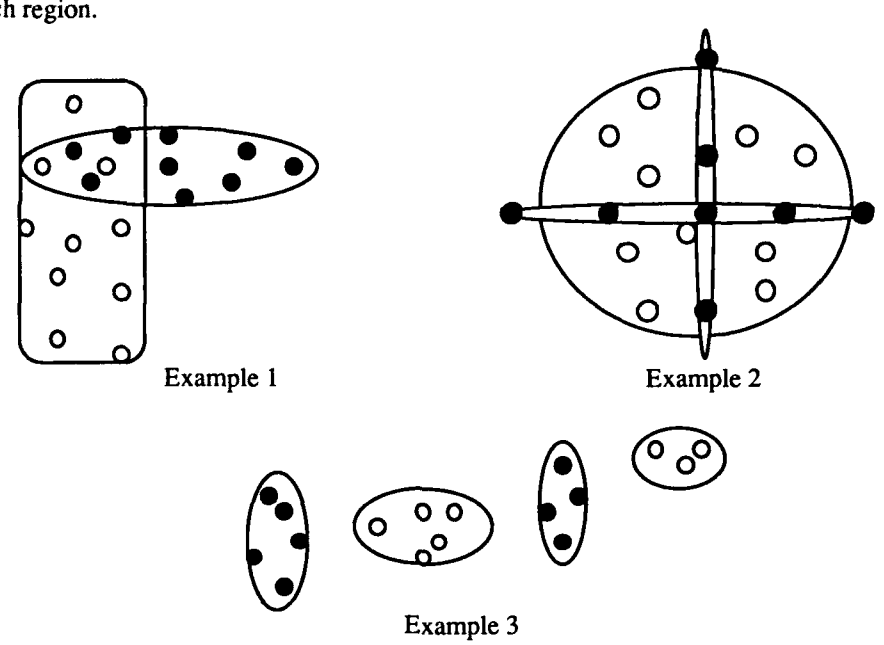


Fig.1: Bayes-Gaussian Classifier Capabilites.
Example 1: Succeeds on two overlapping regions of different classes.
Example2: Fails on thin regions of one class imbedded inside regions of another class.
Example 3: Fails on multiply-connected regions whose mean falls within other regions.

3.2. k-Nearest Neighbor

As in the case of the Bayes-classifier, the success of the k-nearest neighbor classifier can be related to the distribution of the inputs of each class in discriminant space. Since the k-nearest neighbor classifier relies on the category of the nearest

neighbors to classify any vector, problems with regions which are well separated from each other and with a high density of inputs are well suited for k-nearest neighbor classification even if they are multiply-connected. By a high density of inputs, we mean that the variations in the shape of the boundary and the size of the regions is sufficiently greater than the distance between inputs.

Conversely, if the level of noise in the input vector is sufficiently large to cause mixing of input points near the boundaries, or if there aren't enough input points across or inside a region, or along a boundary, then the performance of the k-nearest neighbor classifier will be diminished. We show 2-D examples of appropriate and inappropriate problems for the k-nearest neighbor classifier in Figure 2. Note that if the input vectors are sufficiently sparse or noisy, nearest neighbors may belong to different classes and the classification may fail.

In Example 1, we have repeated Example 3 of Figure 1 to show that multiply-connected, well separated regions can be handled well by the k-nearest neighbor algorithm. The k-nearest neighbor classifier does not attempt to fit a distribution onto the data and only cares about the local distribution. If the local distribution is unambiguous, the classifier can handle multiply-connected regions.

In Example 2, we have presented a noisy boundary where black and white dots are mixed as in Example 1 of Figure 1, but in this case, the result is the reverse. The Bayes-Gaussian classifier sees a global distribution, determines a probability that a point belongs to a particular class and can therefore ignore errors due to noise. But since the k-nearest neighbor classifier only sees local distributions, it may misclassify a white dot inside the white region as a black dot because its nearest neighbor is a black dot (for k = 1). By increasing k, misclassifications due to a "bad" nearest neighbor may be eliminated, but if k is too large, the resolution of variations in the boundary may be lost and "bad" points from the opposite regions may be included.

In Example 3, using a majority rule, each black dot in the long thin region will be misclassified as a white dot since the nearest neighbor of every black dot is white and the majority of any k-nearest neighbors of a black dot are also white. We can let k = 4 or 5 and then use the rule that if one of k neighbors is black, then the dot is classified black. This will succeed in classifying the black sites as black but it will misclassify some white sites as black. Note that if the black dots were sufficiently dense so that most black dots had black neighbors then the k-nearest neighbor classifier would succeed. The difficulties occur when the data is sufficiently sparse either across or within a region so that the nearest neighbors are often of a different class. On the other hand, the Bayes-Gaussian will succeed with this problem. It will choose a tight spread vertically and a broad spread horizontally for the black dots.

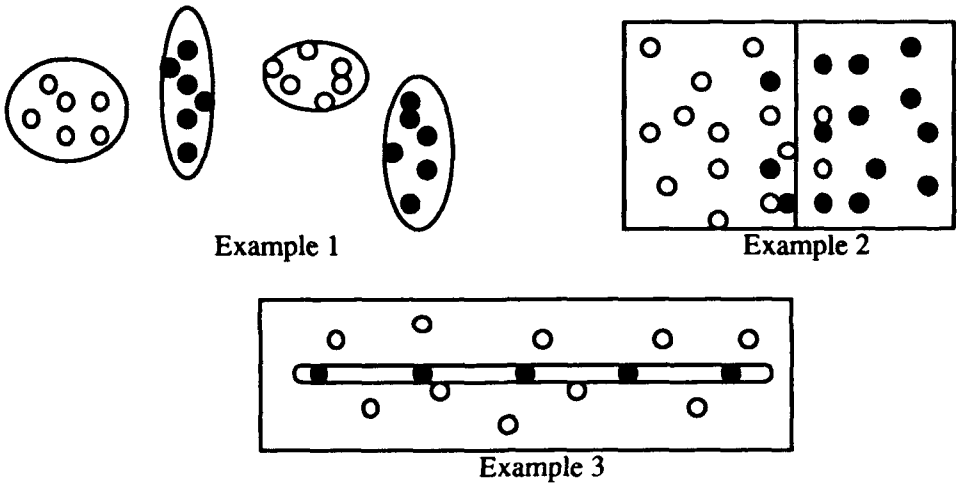


Fig. 2: k-nearest neighbor classifier capabilities.
Example 1: Succeeds with multipy-connected regions where nearest neighbors are always of the correct class.
Example 2: Has difficulty with noisy boundaries when only one nearest neighbor is used.
Example 3: Fails with thin regions where the nearest neigbors belong to the other class.

3.3. Neural Networks

The neural network succeeds with all of the examples in Figure 1 and Figure 2. Each node in the intermediate layer of the network can draw a hyperplane (a line in 2-D) to separate regions of different classes. Given enough nodes, the neural network can form convex or concave, multiply-connected regions with complicated boundaries as needed, and it can do it within the same problem. The example shown in Figure 3 combines concave and convex regions, stringy regions with sparse data and a noisy boundary into the same problem. Both the Bayes-Gaussian and the k-nearest neighbor classifier would fail to varying degrees with different parts of this problem as already described. The neural network will draw a line (hyperplane) at each boundary and it will use two or more lines to fit curved boundaries (like the bottom curve in Figure 3). Care must be taken not to "overlearn" noisy boundaries, otherwise the neural network may draw a complicated curve or form isolated regions around noisy points in the training set, instead of the straight line shown at the noisy boundary in Figure 3. Therefore, with noisy data, it is important not to train the neural network too long.

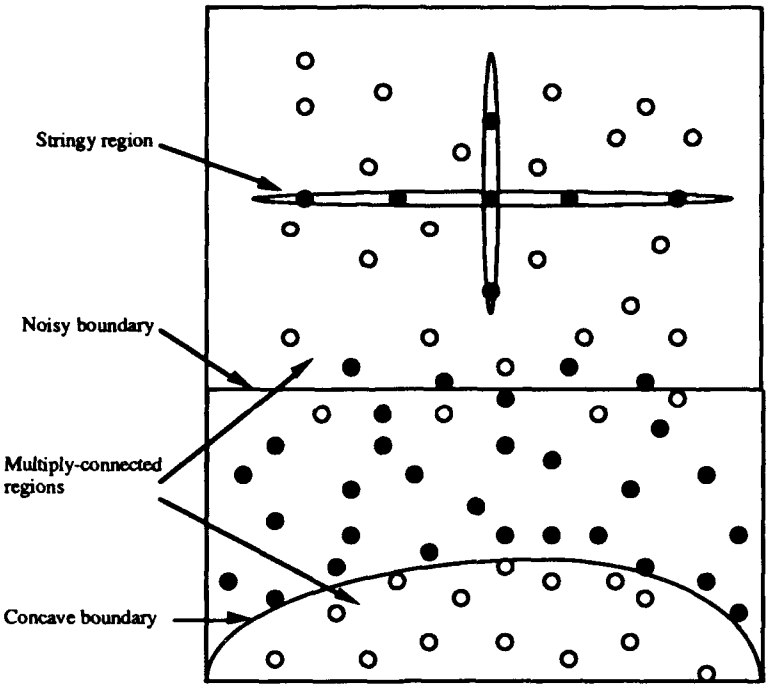


Fig.3: Example of a difficult classification problem which combines features than foil both Bayes Gaussian and k-nearest neighbor classifiers, but can be solved with a neural network.

The particular geometry of the class regions in discriminant space gives a good indication of the number of intermediate levels and nodes needed to solve the problem. More complicated boundaries or a larger number of separate regions will require more hyperplanes and therefore more nodes. The time or the number of learning iterations needed to achieve a sufficiently successful classification also grows with the complexity of the geometry and is discussed in the next section.

4. Time and Storage

4.1. Bayes-Gaussian

Training the Bayes-Gaussian classifier requires the calculation of the averages of each input discriminant, the covariance matrix of each class and their inverses. The main difficulty in implementing the Bayes-Gaussian classifier is in calculating the inverse and the determinant of the covariance matrix C. If many of the elements of C turn out to be near zero or nearly the same, the matrix will be singular or the determinant may be extremely small. In such cases, the covariance matrix can be decomposed, and then inverted approximately. Let n be the number of discriminants and N be the number of points in the training set. Finding the averages of the inputs <x> involves an iteration over the number of inputs n for each point in the training set, a total of nN iterations. For

each training point, the covariance matrix, C, requires iterating over both indeces, or n2 N iterations for the entire training set. If C is nearly singular, then the decomposition process involves 10 n3 iterations. The evaluation of the inverse after the decomposition takes n3 iterations. Altogether, the training time for a Bayes-Gaussian classifier takes on the order of n2 N+n3 iterations with an additional 10 n3 if the decomposition of C is necessary.

Also, this classifier must store the 2n class 1 and class 2 averages of the discriminants, the determinants of the class 1 and class 2 covariant matrices (2 numbers) and the inverse of the covariant matrices which have $2n^2$ elements combined. Therefore, the total storage requirement is $2n^2 + 2n + 2 \approx 2n^2$. During computation it is necessary to store the covariance matrices and their decompositions, of the order of $6n^2 + 0(n)$ numbers.

Classifying a test point x(a) mainly involves the evaluation of the Gaussian probability distribution which requires n^2 iterations to evaluate the products $(x(a) - \langle x \rangle)$ C^{-1} $(x(a) - \langle x \rangle)$ in the exponent of the Gaussian distribution p(x|1). These storage requirements and the number of iterations necessary for training and testing are listed in Table 1 along with the k-nearest neighbor and neural network classifiers.

4.2. k-Nearest Neighbor

The k-nearest neighbor classifier requires no training since it is not reducing the number of training set inputs (e.g., replacing them with a smaller number of prototypes). The algorithm must store the positions in discriminant space of all the input vectors in the training set (n discriminants per input vector) and its desired output (class 1 or class 2) permanently, which requires storing (n + 1) N numbers, where N is the number of points in the training set. Testing is carried out by comparing with all points in the training set. To classify a given point, the k-nearest neighbor classifier must calculate the distance from that point (an n-dimensional vector) to every other point (another n-dimensional vector), which involves nN iterations. Then, these distances must be sorted to find the k shortest distances, which requires kN plus k (k - 1) iterations. Therefore, the total number of testing operations scales with N as (n + k) N + k (k - 1) or $\approx$ (n + k) N for N >> k. For the typically large numbers of inputs, this testing time can become prohibitive. The storage requirements and the number of training iterations are listed in Table 1.

4.3. Neural Network

After the learning process, the neural network stores all its information in the weights between the nodes, and the thresholds of non input nodes. The input layer has n nodes corresponding to the n-dimensions of each input vector. For a three layer network we let the second and third layers have n' and n" nodes, respectively. In a network with connections only between adjacent layers, there will be a total of nn' + n'n" weights and n' + n" thresholds to store. Typically, for problems in which many inputs are categorized into a few classes, n' < n and n" << n, which makes this storage requirement smaller than the Bayes-Gaussian storage needs, i.e.,

$$nn' + 0(n', n'', n'n'') < n^2 + 0(n), \qquad n > n'$$

and it is smaller than the k-nearest neighbor storage requirement for large N, i.e.,

$$nn' + 0(n', n'', n'n'') < nN, \qquad N > n' .$$

The learning time of the neural network is difficult to estimate. With a standard back-propagation network, at the introduction of each input, $nn' + n'n''$ weights and $n' + n''$ thresholds must be modified. If it takes g iterations over the entire training input N to reduce the error below a desiredvalue, then the learning time will scale approximately as

$$gN(nn' + n'n'').$$

However, the value of g depends on the particular problem and there is currently no general method for determining g. The neural network may take longer to train than the Bayes-Gaussian, i.e.,

$$gN(nn' + n'n'') > 10n^3 + 0(n^2)$$

$$\text{if } gNn' > 10n^2,$$

but this is not necessarily true.

5. Conclusions

Through a geometrical representation of the categorization problem, we have described and illustrated general types of problems that can be handled by various classifiers. If objects of different classes, represented by vectors in discriminant space, separate into regions, then discrimination is theoretically possible. The Bayes-Gaussian classification scheme is suited to problems in which each class of objects separates in discriminant space into a single, convex region (with or without noisy boundaries). It is not successful with problems in which a class of objects is represented by multiply-connected regions or if the regions are concave, as shown in example 2 of Fig. 1.

On the other hand, the k-nearest neighbor classifier succeeds with problems containing concave regions or multiply-connected regions corresponding to one class, but it may fail if it encounters noisy boundaries between classes or stringy regions or sparse data across a region in discriminant space, as illustrated in Fig. 2.

The superior performance of the neural network results from its versatility: It does not assume any particular distribution of the inputs as does the Bayes-Gaussian classifier, nor does it assume that proximity in discriminant space determines the class of an object as does the k-nearest neighbor classifier. Instead, the neural network draws boundaries between regions by combining a sufficient number of hyperplanes to accurately match all boundaries. To do this, it requires some learning time, during which it adjusts the positions and orientations (thresholds and weights) of the hyperplanes in order to best match the given boundaries between all regions.

Furthermore, complicated problems which may contain many different kinds of regions and boundaries, such as that shown if Fig. 3 (especially if they are unknown a priori), can only be handled by a versatile classifier which is able to design different regions and boundaries to suit different parts of the problem.

Also, in considering storage and time, we have shown that the k-nearest neighbor classifier's storage and time requirements scales with N, the number of inputs. For large N, it is advantageous to store a reduced set of numbers rather than the entire input set to avoid this problem, which both the Bayes-Gaussian and neural network classifiers do. On the other hand, the storage requirement and matrix inversion calculation time of the Bayes-Gaussian classifier scale as n^2 and n^3, respectively, where n is the number of discriminants. This may become prohibitive if n is large. The neural network only needs to store the weights and thresholds. Therefore, its storage requirement does not scale with the number of inputs and its time requirement does not necessarily scale with n^2. (In some problems, the learning time may be shown to scale linearly with n.) It is difficult to give a general expression for the learning time, since it depends on the number of iterations over the input set necessary to converge to an acceptable solution, g, which is usually not known a priori.

Since the performance of the neural network always matches or surpasses that of either the Bayes-Gaussian or the k-nearest neighbor classifiers, and its storage and time requirements are often lower for the static classification problem, it is almost always the best choice among these three classifiers, particularly for complicated problems or problems with large inputs and large numbers of discriminants.

Fuzzy Logic and the Relation to Neural Networks

Dr. Gregg Wilensky

September 7, 1990

Research sponsored by DARPA and ONR under contract N00014-89C-0257.

1. Introduction

Fuzzy logic is a semi-empirical description of the application of logical reasoning to ill-defined sets. Its utility lies in the ability to impose a logical structure without requiring the strictness of logical truth and falsity. Let me clarify this by giving an example based on a recent application by Sony Corp. of a fuzzy logic expert system for optimization of the image on an advanced television monitor[16]. 248 regions on the image are monitored 60 times a second. Using prerecorded fuzzy logic rules, changes are made in picture contrast, hue, color saturation and detail for each of the regions. As TV reception changes, the modifications compensate to get the "best" possible picture. In addition, because the image is monitored and modified locally, good detail can be maintained in one portion of the image while another portion is smoothed out to get rid of speckle (in a portion of the sky for example). Although the details of the algorithm have not been released, one can envision how it might work. For each region of the image sampled, one has a number of image pixels, their intensity and color values. Every such image is ranked into various categories or sets. Example sets might be Sky-like, Background, Detailed, as well as more general categories such as Noisy, Smooth, Interesting, Monotonous, Bright, Dim. The logic of the Advanced Signal Correction system would then include statements such as: If Monotonous and Noisy then smooth out the image, decrease contrast. If Sky-like and Noisy and Dim then do the same and increase blueness. If Face and Smooth and TVisOld then increase contrast 20% else if Face and Smooth and TVisNew then increase contrast 10%.

These concepts such as Smooth, Noisy, etc. are not well defined. In "Fuzzy" terminology they are said to be fuzzy sets. What distinguishes a fuzzy set from an ordinary set is that a fuzzy set can have partial membership. In ordinary set theory, an element is either a member of a particular set or it is not. In fuzzy set theory, an element can be a 30% member, for example. Every element is given a membership number for every set which ranges from 0 to 1. 0 corresponds to "not a member" and 1 corresponds

---

[16]R. Doherty, *Electronic Engineering Times*, April 30, 1990, pg. 4.

to "definitely a member". Anything in-between is a partial member. Now what does partial membership mean anyway? Consider the set of noisy images. To calculate whether a given image should lie in the set one might calculate the mean squared deviation of all the pixel intensities from the mean intensity. Let's call the square root of this number divided by the mean intensity the noise-to-signal ratio ν. If we were working with ordinary set theory we might say that the image lies in the Noisy set if ν is greater than some number, let's say 0.7. Now this number is quite arbitrary. After all, who's to say that 0.6 or 0.5 are not almost as good cutoffs? Or more importantly, for the task of adjusting the television set, perhaps the set should be adjusted slightly if ν is greater than 0.5, even more if ν is greater than 0.6 and even more if ν is greater or equal to 0.7. Now this could be done by simply including a lot of if-then statements: if the value of ν lies in such and such a range then adjust the television parameters by such and such an amount. But then the simple logical rules that one started with (if the image is noisy then adjust the set) become more and more fragmented. It would work just as well, its just more complicated. Another alternative is to find a function $f(\nu)$ which determines the adjustment needed. This works just as well but now the logical rules are obscured. The advantage of the fuzzy set approach is that the logical rules are specified (so that a human can maintain control over and understanding of the TV operation) and yet a range of possibilities is allowed; one does not need a bunch of if-then statements to represent the concept of noisiness.

The membership value of an element in a fuzzy set is interpreted as the possibility that the element belongs to the set. It is distinct from a probability[17]: probability refers to an ensemble of elements each one of which either does or does not belong to the set. By counting the number of elements which do belong to the set one obtains the probability that an element of the ensemble lies in the set. On the other hand, with a fuzzy set, a given element does not necessarily lie in or outside of the set. Any single element has a possibility of lying in the set and a possibility of lying outside the set. The uncertainty in these possibilities is the uncertainty in the definition of the set rather than the uncertainty of the outcome of a given event or element of the set.

The utility of describing fuzzy sets lies in the ability to describe logical operations on the sets which make reasonable sense and which reduce to ordinary logical rules of non-fuzzy sets when the sets are indeed not fuzzy (i.e. they consist only of elements which are either 0 or 1). As an example, consider the union of two sets DS and LE, dark-skinned and light-eyed individuals. These concepts are not rigidly defined and so are prime candidates for fuzzy sets. If these were ordinary sets, the intersection DS & LE would consist of all individuals who have both dark skin and light eyes, i.e. the overlap between the two sets of well-defined elements, the set of elements which have membership equal to unity in both set DS and set LE. For fuzzy sets, a modified definition is needed and is best expressed in terms of the element membership or possibility functions. $p_1(x)$ is the possibility that the element x lies in the set DS, and similarly $p_2(x)$ is the possibility that element x lies in the set LE. These are numbers which lie between 0 and 1. The possibility function $p_{1\ \&\ 2}(x)$ for the intersection of the two sets is defined (somewhat arbitrarily) as the minimum of the two possibilities:

---

[17]There are a lot of arguments in the literature about whether this concept of possibility is really different from probability. A discussion is given in the following reference:
E. H. Mamdani, B. R. Gaines, ed., Fuzzy Reasoning and its Applications, 1981, Academic Press, N.Y., pg. 8.

$$p_{1\ \&\ 2}(x) = \min(\, p_1(x),\ p_2(x)\,).$$

For ordinary sets, $p_1(x)$ and $p_2(x)$ are either 0 or 1 for every element x. Then the minimum is unity only for those elements which are both members of set 1 and members of set 2; i.e. this definition reduces to the standard definition of the intersection for ordinary sets. In a similar manner, the possibility function for the union of two fuzzy sets is defined as the maximum:

$$p_{1\ +\ 2}(x) = \max(\, p_1(x),\ p_2(x)\,).$$

These definitions are illustrated in Figure 1. While these choices for fuzzy set intersection and union are simple and reduce to the proper definitions for non-fuzzy sets, they are not the only possible definitions that work. Another possibility for the intersection is $p_{1\ \&\ 2}(x) = p_1(x) * p_2(x)$ for example. The definitions can be extended to more than two sets. For example,

$$p_{1\ +\ 2\ +\ 3}(x) = \max(\, p_1(x), p_2(x), p_3(x)\,).$$

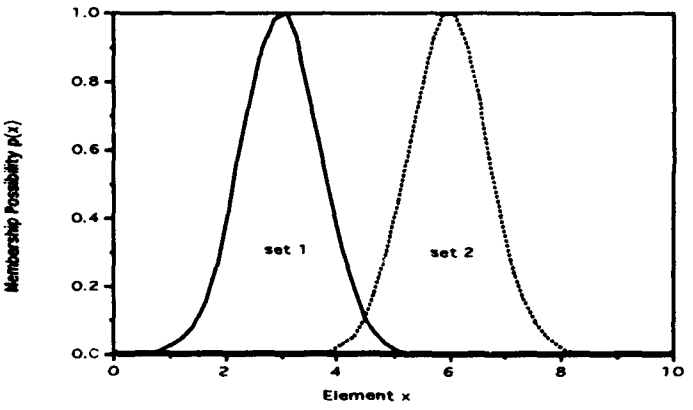


Fig. 1-a

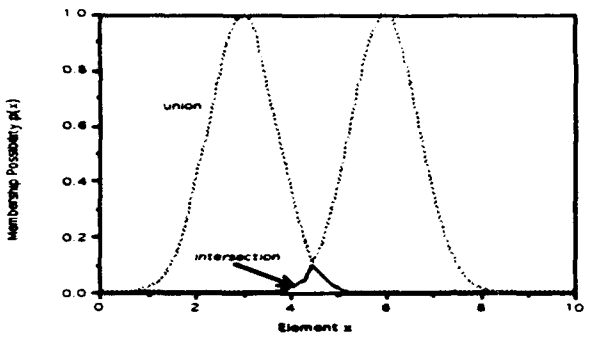


Fig. 1-b

Figure 1: Example possibility functions. Fig 1-a: Examples for two fuzzy sets. Fig 1-b: Examples for the intersection and union of the fuzzy sets shown in Fig. 1-a.

Another concept which is needed for fuzzy sets in order to quantify a statement such as "if one has light skin then one is likely to have dark eyes" is the notion of implication. A implies B (shorthand, $A \rightarrow B$) means that if A is true then B is true. That is not to say that A is necessarily causally related to B, only that A and B are correlated. In the ordinary crisp logic this is translated into the following truth table:

p_A	p_B	$p_{A \rightarrow B}$
0	0	1
0	1	1
1	0	0
1	1	1

In other words, the statement $A \rightarrow B$ is taken to be true unless A is true and B is false. Since the truth table is reproduced if one equates $p_{A \rightarrow B} = \min(1, 1 + p_B - p_A)$, this equation is often extended to fuzzy sets to give the possibility that A(x) implies B(x'). As an example, consider the proposition mentioned above: "light skin implies dark eyes". Let set 1 be the set of light skinned people. It will be defined in terms of a reflectance measurement which gives a value x for each person. The possibility function $p_1(x)$ then defines what we mean by light-skinned. It may be a Gaussian distribution centered about a reflectance of 0.7, for example. Dark-skinned could be represented by a possibility function which is a Gaussian distribution centered about a reflectance of 0.3. For most concepts of this type these distributions are not well defined. We have a rough idea of what they should look like, but can't really quantify them other than by guessing. In a similar manner we will take the possibility function $p_2(x')$ to represent dark eyes, where x' is some quantifiable measure of eye color intensity. Given a person with skin tone x we can estimate the possibility that he or she will have eye color intensity x' as: $\min(1, 1 + p_2(x') - p_1(x))$.

While the above truth table is standard logic it has problems. The two situations where there is no information as to the truth of the implication (the first two rows) are given truth values of 1. In ordinary logic this is natural; the value has to be either 0 or 1 and it is not necessarily false. But a better representation is available with fuzzy logic. We can assign a possibility of 1/2 to these situations to represent the fact that we simply have no information. Thus a truth table which more accurately reflects our knowledge would be the following:

p_A	p_B	$p_{A \to B}$
0	0	0.5
0	1	0.5
1	0	0
1	1	1

One could translate this into a formula such as

$$p_{A(x) \to B(x')} = \min\left[1, 1 + p_{B(x')} - p_{A(x)}\right] - 0.5\left[1 - p_{A(x)}\right].$$

For the application of fuzzy logic television adjustment, such formulae can be used to take an input image represented by x and determine the output adjustment x'. More complicated logical expressions can be built with combination of "and", "or" and "implication".

2. An Example; the Fuzzy Washing Machine:

To demonstrate the approach an example of a fuzzy washing machine controller will be shown using a plausible set of rules. The washing machine will be assumed to be capable of measuring and adjusting water temperature, wash-cycle time, detergent concentration and water murkiness. Each of these will be assumed to be described by fuzzy sets. For example, water temperature may be cold, lukewarm or hot (finer gradations such as very hot, slightly hot may be useful but will not be considered here for simplicity) Thus, if T is the water temperature at a given time, then $p_{hot}(T)$ describes the possibility distribution for the hot water set. Similarly, $p_{lukewarm}(T)$ and $p_{cold}(T)$ describe the lukewarm and cold water sets. Water cleanliness may be measured by a light transmission measurement which gives a value τ for the transmissivity and we will consider two fuzzy sets, clean and dirty, described by the possibility functions $p_{clean}(\tau)$ and $p_{dirty}(\tau)$. The wash time t will be described in terms of the sets long, medium and short: $p_{long}(t)$, $p_{medium}(t)$, $p_{short}(t)$. Detergent concentration n will be described as strong or weak: $p_{strong}(n)$, $p_{weak}(n)$. And finally, the whiteness w of the load will be described by $p_{white}(w)$ and $p_{dark}(w)$. These fuzzy sets are summarized below:

water temperature: T	hot	lukewarm	cold

water cleanliness: τ	clean	dirty	
wash time: t	long	medium	short
detergent concentration: n	weak	strong	
clothes whiteness: w	white	dark	

Consider the following set of fuzzy logical rules made up by a supposed expert in clothes washing:

if (the water is dirty and the detergent concentration is weak) then
change the detergent concentration to strong

if (the water is dirty and the detergent concentration is weak and the wash time is short) then
change the wash time to long

if (water temperature is cold and clothes are white) then
change water temperature to hot
or
if (water temperature is hot and clothes are dark) then
change water temperature to cold.

In an obvious shorthand notation, this could be written as

if (dirty & weak) then strong

if (dirty & weak & short) then long

if (cold & white) then hot
or
if (hot & dark) then cold.

These statements need to be translated into a form involving the possibility functions so that the controller can adjust the settings for water temperature, wash time, and detergent concentration. This is done in the following manner (different authors use slightly different variations of the approach shown here):

Given a set of input measurements T, τ, t, n and w and given the above set of fuzzy logical rules,we wish to find the desired output values for T, t, and n; that is, we would like to adjust the water temperature, the wash time and the detergent concentration. Both the input parameters and the output settings are described by fuzzy sets. There is a possibility for each value of the output parameters. To calculate the actual value to set for the temperature, for example, we will calculate the possibility function for T given the inputs and the logical rules. The value of T when averaged over the possibility function could provide the desired output value:

$$T_{output} = \langle T \rangle = \left[\int dT' \, p(T') \, T'\right] / \left[\int dT' \, p(T')\right] .$$

An alternative form

$$\Delta T_{output} \propto [\int dT' [p(T')-1/2]\, T']/ [\int dT' p(T')] \ ,$$

gives the change in temperature such that when there is no knowledge, i.e. when p(T') = 1/2, there is no change. The term '-1/2' subtracts out a median temperature. p(T') is the possibility that the water temperature should be set equal to the value T' given the set of input parameters and the set of fuzzy logical rules. The other parameters are treated in a similar manner. Our objective then is to find the possibility functions p(T'), p(t'), and p(n') given the set of input measurements and the set of logical rules. A prime is used here to distinguish the input parameters(unprimed) from the output variables. We first reduce the calculation to the basic set of implication possibilities p[A&B→ C] (the possibility that if A is true and B is true then C is true) by recognizing that the 'or' statement which ties together the last two 'if-then' statements can be represented by taking the maximum of the possibility functions for each 'if-then' statement. This leads to the following:

$$p(n') = p[\text{dirty}(\tau) \,\&\, \text{weak}(n) \rightarrow \text{strong}(n')]$$

$$p(t') = p[\text{dirty}(\tau) \,\&\, \text{weak}(n) \,\&\, \text{short}(t) \rightarrow \text{long}(t')]$$

$$p(T') = \max[\, p[\text{cold}(T) \,\&\, \text{white}(w) \rightarrow \text{hot}(T')],\ p[\text{hot}(T) \,\&\, \text{dark}(w) \rightarrow \text{cold}(T')]\,].$$

The basic possibility functions in turn can be expressed as

$$p[A \,\&\, B \rightarrow C] = \min[1,\, 1 + p_C - p_{A\,\&\,B}] - 0.5[1 - p_{A\,\&\,B}] \ , \text{ where}$$

$$p_{A\,\&\,B} = \min(p_A, p_B).$$

Thus,

$$p(n') = \min[1,\, 1 + p_{strong}(n') - \min(\, p_{dirty}(\tau),\, p_{weak}(n)\,)] - 0.5[1 - \min(\, p_{dirty}(\tau),\, p_{weak}(n)\,)],$$

$$p(t') = \min[1,\, 1 + p_{long}(t') - \min(\, p_{dirty}(\tau),\, p_{weak}(n),\, p_{short}(t)\,)] - 0.5[1 - \min(\, p_{dirty}(\tau),\, p_{weak}(n),\, p_{short}(t)\,)],$$

$$p(T') = \max\{\ \min[1,\, 1 + p_{hot}(T') - \min(\, p_{cold}(T),\, p_{white}(w))]$$

$$- 0.5\left[1 - \min(p_{cold}(T), p_{white}(w))\right], \min\left[1, 1 + p_{cold}(T')\right.$$

$$\left. - \min(p_{hot}(T), p_{dark}(w))\right] - 0.5\left[1 - \min(p_{hot}(T), p_{dark}(w))\right] \Big\}.$$

Using the above expression, p(n') can be calculated over the range of values n' that the detergent concentration can take on. The average value of n' over this possibility distribution minus the median concentration gives the change in concentration for the new setting. Consider the case in which the transmissivity τ is small so that the water is very dirty and the detergent concentration n is small so that the concentration is very weak. Then the possibilty that the water is both dirty and weak will be large. The possibility function for the implication dirty&weak $\rightarrow$ strong(n') will then follow the possibility distribution strong(n'). If n' is small, the concentration is weak and the truth of the implication will be small. If n' is large, the concentration is strong and the truth of the implication will be large. For this situation then $p(n') \approx p_{strong}(n')$ in a rough sense. The average value of n'would then correspond to a characteristic strong concentration. The concentration would be changed to this value if the first representation described above is used. If the second representation is used then n' would be changed by an amount proportional to the difference of the characteristic strong temperature from a median temperature. On the other hand, when the transmissivity is large or the concentration is large, the truth of the implication will be set close to 1/2, independent of the value of n'. There is no knowledge about the implication and the output value of n' will be unchanged: $\Delta n' \approx 0$ if the second representation is used. If the first representation is used n' will be set to a median concentration. In a similar manner the change in the wash time and the water temperature can be calculated and adjusted to the average possible value.

Given a reasonable set of rules for controlling the washing machine, the next step is to better define the fuzzy sets by giving the possibility functions a specific functional form. Typically one chooses a simple function with a few adjustable parameters. An example would be a Gaussian function for the lukewarm water temperature distribution:

$$p_{lukewarm}(T) = p_0 \exp\left[-(T-T_0)^2/2\sigma^2\right].$$

The distribution function for hot and cold water temperature might take the forms

$$p_{hot}(T) = p''_0 / \left[1 + \exp\left[-(T-T_{hot})/\sigma'\right]\right],$$

$$p_{cold}(T) = p''_0 / \left[1 + \exp\left[+(T-T_{cold})/\sigma''\right]\right].$$

These examples are shown in Figure 2 below.

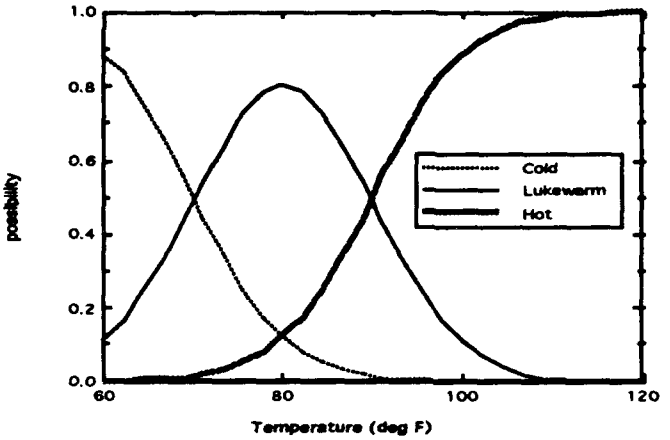


Figure 2: Example possibility distribution functions for water temperature fuzzy sets.

The difficulty in this stage is in choosing good functions and parameters for all the fuzzy sets. It typically involves a lot of guesswork and trial and error testing. One would choose a set of parameters for all the distribution functions and then run the washing machine (or a simulation of the washing machine) to evaluate how well the clothes come out. If something is not right one would try to figure out why and try adjusting the fits to correct it. Running the machine involves repeated sampling of the water temperature and cleanliness, clothes whiteness, detergent concentration and wash cycle setting. For each sampling, output values are calculated for detergent concentration, water temperature and wash time and these are adjusted.

Neural networks offer a great advantage at this point. For control situations in which one can quantify the performance (a simulation of the washing machine might quantify the cleanliness of the clothes after washing, for example), the neural net can be used to optimize the choice of functions and parameters. A schematic example is illustrated in Figure 3 which captures the spirit though not the details of the fuzzy logic rules. The feedforward neural net (which can be trained with the back-propagation algorithm) is used to connect each parameter with each node describing a fuzzy set. These mapping subnets thus have one input and one output node and provide a functional description of the possibility distributions. For example, one subnet will have an input node which represents water temperature and an output node whose value is the possibility function for the 'hot' fuzzy set. This subnet performs the mapping from T onto $p_{hot}(T)$. A separate subnet will map T onto $p_{cold}(T)$. These subnets can be pretrained to provide a rough estimate of the possibility functions, such as the Gaussian function discussed above. They can then automatically be optimized by training the whole network to minimize an error function. In the washing machine example, one might train the net based upon simulations of desired wash cycles.

To complete the whole neural network yet another subnet could take as input all the appropriate fuzzy subset nodes (the outputs of the mapping subnets) and implement the fuzzy logical rules which have been decided upon. There are several possible ways of implementing this. One example is shown in Figure 3 in which the input possibility functions are distinct from the output functions.

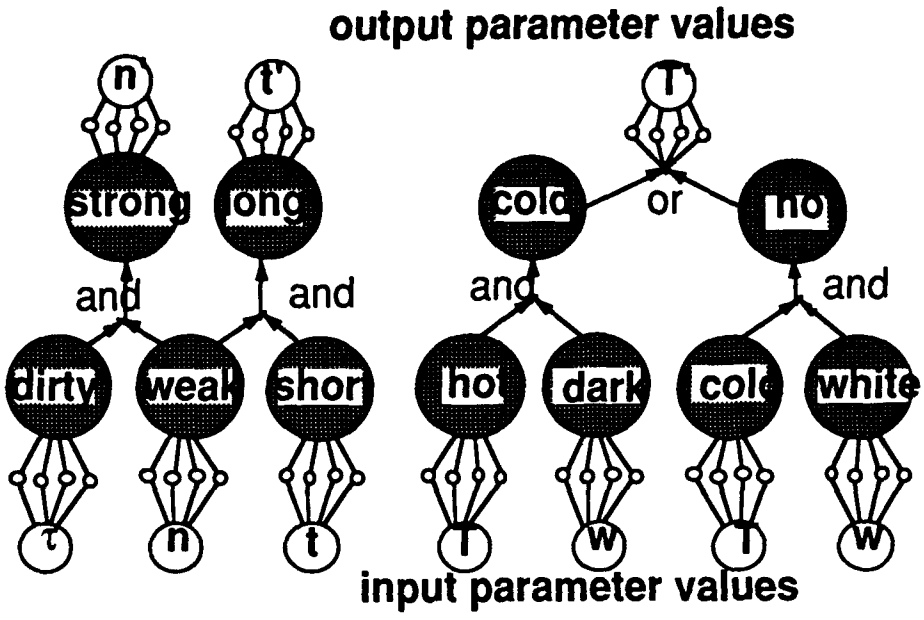


Figure 3: Example of an integrated fuzzy logic neural network. The circles represent neural network nodes. For the input nodes the nodal values are the input values. For the output nodes the nodal values give the recommended values for these control parameters. The large shaded nodes represent the possibility distribution functions for the fuzzy sets. The neural subnets in the lower layer map input parameters to fuzzy set possibilities. Additional subnets are shown which implement the fuzzy logic rules 'and' and 'or'. The top subnets determine the output values in terms of the fuzzy set possibilities.

By training the whole network, both the input and the output mappings can be optimized. In addition, one could allow variation of the network parameters which implement the logical functions 'and', 'or', and 'implies'. The degree to which these remain unchanged indicates the validity of the logical rules. The network performance could be compared with a fully interconnected network to assess the extent to which the fuzzy logic rules achieve a good solution to the problem.

The same set of fuzzy rules shown above could be approximately implemented by an expert system approach in which ordinary logical rules are used. But the same set of rules that could be described by three simple equations for the fuzzy possibility distribution would require implementation of many rules of the form: if ($0.3 < \tau < 0.4$ and $0.5 < n < 0.55$) then $n' = 0.3$. Finer and finer degrees of distinction require more and

more rules. Nor does this approach allow one to vary a few parameters to optimize the solution as can be done with fuzzy logic or a neural network.

3. Summary

The popularity of the fuzzy approach lies in the fact that it is a simple method of extending the expert system approach (specify a bunch of if-then statements to describe the problem) to the real world situation for which the ideas (objects or sets) used are not well defined. Togai InfraLogic Inc. (TIL) of Irvine, California has developed a fuzzy focussing system for Canon 8mm camcorders[18]. Seven input variables such as the frequency distribution of the CCD signal, and its derivative are used; if the image is becoming more focused there will be more high frequency structure in the CCD distribution. Using 20 fuzzy rules, the system produces one output: the focus level. A conventional expert system approach would require approximately 340 rules. The fuzzy program is packed in 2 kbytes of read-only memory and is used to check the focus 30 times a second. A prototype system was developed in less than 7 days. But it took considerably more effort by 3 experts using trial-and-error determination of the possibility functions for the input variables. Another application by the same company, produced for Mitsubishi Heavy Industries Inc., was a fuzzy controller for large office building heating and cooling. The system uses 25 fuzzy rules for heating and 25 for cooling. The inputs include room and wall temperature and their derivatives in time. The initial rules were written in 3 days. The initial possibility functions were generated by experts in about a month and another 3 months were needed to tune the system and optimize performance. The system reduced heating and cooling times by a factor of 5 and improved stability by a factor of two (whatever that really means).

The above examples point out the utility of the fuzzy logic approach and the speed and ease of generating a useful set of logical rules. The drawback of the approach is the large amount of time needed by experts to "guess" good possibility distribution functions and to fine-tune the system. Here is a prime role for neural networks which offer the ability to automatically generate the possibility distributions and to fine tune the system through back-propagation error minimization. A combined fuzzy logic/ neural net approach could offer the benefits of having an imposed logical structure and an optimized system with no need for the devotion of a lot of time by experts.

In summary, fuzzy logic is seen to offer the advantage of building-in logical relations among fuzzily defined sets or ideas. Having defined a set of logical relations, one can then calculate a mapping from a set of input parameters to a set of output parameters which is in some fuzzy way based on the logical relations. This should be contrasted with the typical supervised neural net approach in which one determines the relation between inputs and outputs so that one minimizes the error made on some set of data. The fuzzy logic approach offers a fuzzy understanding of the logic rules involved (they are put in by hand) but does not guarantee that this is the best solution that minimizes the error made. On the other hand, the neural net approach does guarantee

[18]T. J. Schwartz, "Fuzzy Systems in the Real World", *AI Expert*, August 1990, pgs. 29-36.

error minimization but does not provide a simple understanding of any overlying logical rules which might generate the mapping. They are complementary approaches. A natural extension of both methods is a unified fuzzy neural network in which one builds the logical operations as neural net components and then trains the net. One can test the validity of the logical constructs by observing whether they change as the net is trained. If the logical operations are still valid after training then one has a network in which one understands the fuzzy logical structure and yet still minimizes the error. In addition, the neural network can be used to determine the possibility distributions directly rather than the usual method of trial-and-error guessing. These ideas are described further in reference 3. Although there are several papers in the literature combining fuzzy logic with neural nets, the ability to test and refine the logical rules remains to be implemented.

One might ask: why bother with fuzzy rules? why not just use a neural network or a more standard control system? The use of fuzzy rules do offer one clear advantage, apart from the emotional impact of simply allowing a human to understand what the system is doing: new rules could be added relatively easily without having to completely redesign and retrain the neural network. A demonstration is needed of these advantages of a combined fuzzy logic neural network.

The Reduced Coulomb Energy (RCE) Classifier

February 26, 1990

1. RCE Supervised Learning Algorithm (In a Nutshell)

Consider the 2-D input space pictured in Figure 1A, where each input vector is represented by a dot: a black dot is an input belonging to class 1, and a white dot is an input belonging to class 2. The basic learning procedure of the RCE algorithm consists of creating prototype vectors of each class which coincide with some input vectors of that class, building hyperspheres of that class centered around each prototype, and then shrinking the hyperspheres until only inputs of the correct class are within each hypersphere. In other words, hyperspheres are created and shrunk until all class 1 inputs are covered by only class 1 hyperspheres and all class 2 inputs are covered only by class 2 hyperspheres. (The final picture is shown in Figure 1B.)

2. RCE Algorithm (More Precisely)

Begin with a set of input vectors $\mathbf{x}_i$ where each component of the vector corresponds to some discriminant that may be useful in determining the appropriate class C_i of the input vector. Select the training set from these input vectors $\mathbf{x}_i$ and their corresponding correct classes C_i. Define a set of "prototype" vectors $\mathbf{p}_j$, as yet undetermined in number and value, which will serve to define the regions of each class. Choose an initial radius r_0.

Let the first prototype $\mathbf{p}_1$ equal the first input vector $\mathbf{x}_1$, $\mathbf{p}_1 = \mathbf{x}_1$, and let the first "hypersphere" h_1 have its center at $\mathbf{p}_1$ and radius $r_1 = r_0$. Next, introduce each input vector $\mathbf{x}_i$ with its corresponding classification C_i one at a time. For each i, search for all hyperspheres $h_j$which enclose $\mathbf{x}_i$, i.e., all h_j such that

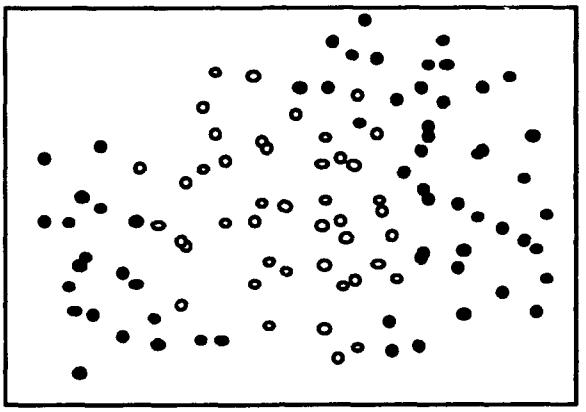
$$|\mathbf{x}_i - \mathbf{p}_j| < r_j .$$

Each hypersphere h_j which encloses $\mathbf{x}_i$ corresponds to some class C_j:
If $C_i = C_j$, do nothing.
If $C_i \neq C_j$, shrink hypersphere h_j until $\mathbf{x}_i$ lies on the sphere, ie:reduce r_j until $|\mathbf{x}_i - \mathbf{p}_j| = r_j$.

If no h_j encloses x_i, then create a new prototype and its corresponding hypersphere by setting it equal to x_i, i.e.,

$$p_k = x_i .$$

Continue this process of shrinking r_j and creating new prototypes p_k until each input is categorized in its correct class. This algorithm is illustrated in Figure 2.



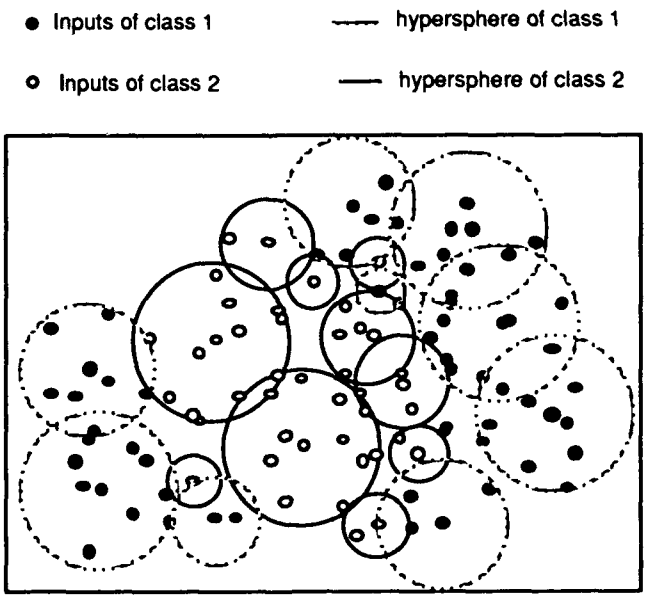


Fig. 1: Illustration of the RCE learning algorithm:

A) 2-dimensional input vectors are represented as black dots if they belong to class 1 and black dots if they belong to class 2.

B) RCE creates circular prototypes centered about sample input points and then shrinks them until only one class is included within the prototype radius.

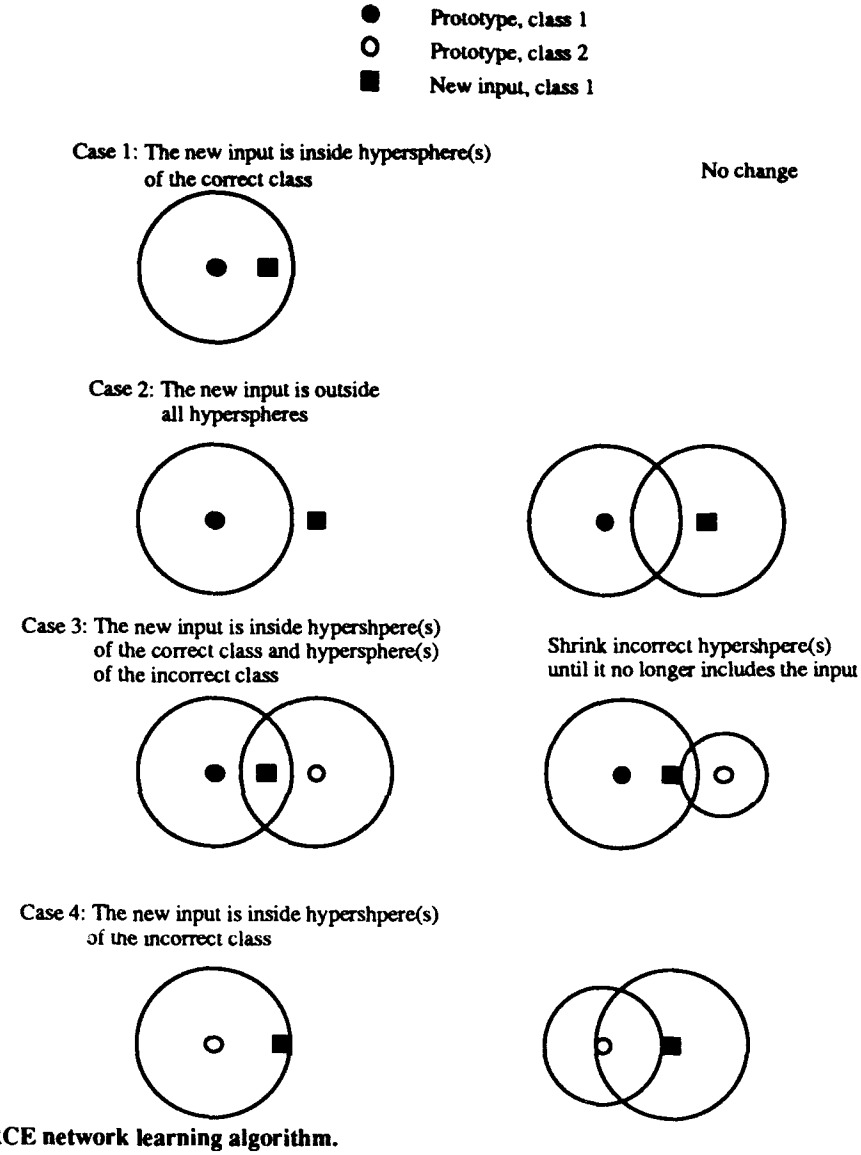


Fig. 2: RCE network learning algorithm.

Note that:

1. More than one prototype may represent a class.
2. All of space is not necessarily enclosed by hyperspheres, some gaps may remain.
3. Hyperspheres of the same class can overlap.
4. Hyperspheres of the opposite classes may overlap if none of the inputs presented during training fall in the overlapping region.

3. Neural Network Implementation

To construct a neural network RCE classifier, we let the input nodes (1st layer) correspond to the components of the input vectors $\mathbf{x}_i$ and assume that they are normalized, i.e.,

$$|\mathbf{x}_i| = 1.$$

Let the weight vector $\mathbf{A}_j$ to each node j in the intermediate (second layer) correspond to a normalized prototype $\mathbf{p}_j$ times a parameter $\lambda_j = \lambda_0$

$$\mathbf{A}_j = \lambda_j \mathbf{p}_j.$$

Since the vectors $\mathbf{x}_i$ and $\mathbf{p}_j$ are all normalized and lie on an n-dimensional hypersphere, the distance between any two vectors can be measured by the angle between the vectors, φ, which is related to the dot product of the two vectors:

$$\mathbf{x}_i \cdot \mathbf{p}_j = |\mathbf{x}_i| \, |\mathbf{p}_j| \cos \varphi = \cos \varphi .$$

The input to the jth neuron of the second layer is:

$$\begin{aligned} \mathbf{A}_j \cdot \mathbf{x}_i &= \lambda_j \mathbf{p}_j \cdot \mathbf{x}_i \\ &= \lambda_j \cos \varphi , \end{aligned}$$

and is a measurement of the distance between the input and the prototype j times the parameter λ_j.

In order to determine whether or not $\mathbf{x}_i$ is in the hypersphere h_j, in the RCE algorithm, we compared the distance between $\mathbf{p}_j$ and $\mathbf{x}_i$ to the radius r_j. In the neural network, we can compare $\cos \phi$, which is a measure of the distance between $\mathbf{p}_j$ and $\mathbf{x}_i$ to a radius r_j, or equivalently, we can compare $\lambda_j \cos \phi$ to a constant threshold θ. Let the output of the jth intermediate layer neuron be

$$\begin{aligned} &\mathbf{A}_j \cdot \mathbf{x}_i && \text{if } \mathbf{A}_j \cdot \mathbf{x}_i > \theta \\ &= 0 && \text{if } \mathbf{A}_j \cdot \mathbf{x}_i \leq \theta . \end{aligned}$$

The output of a second layer neuron, which represents some prototype, indicates whether or not a particular input lies within the hypersphere of that prototype. If the jth neuron prototype is of the incorrect class and its output is non-zero, in the RCE algorithm, we reduced r_j to shrink the hypersphere h_j; in the neural network, we reduce λ_j until $\mathbf{A}_j \cdot \mathbf{x}_i$ is equal to θ and the output of the jth neuron is zero. Note that since prototypes can have overlapping hyperspheres, more than one neuron on the second layer can have a non-zero value.

Olfactory Cortex

In the output (third) layer, each neuron represents a single class C_i. It is connected only to prototypes of that class. Thus, the second and third layers are not fully connected. In other words, all prototypes of class 1 are connected to the class 1 output neuron only, all prototypes of class 2 are connected to the class 2 output neuron, and so on.

Initially, there are no prototypes or weights. As a prototype is created, it is connected to the appropriate class output and the connecting weight is set equal to 1. There is no competition or inhibition between output cells. Their values are either zero or non-zero; a single prototype has as much effect as many prototypes in that it fires the corresponding output neuron. This neural network architecture is illustrated in Figure 3.

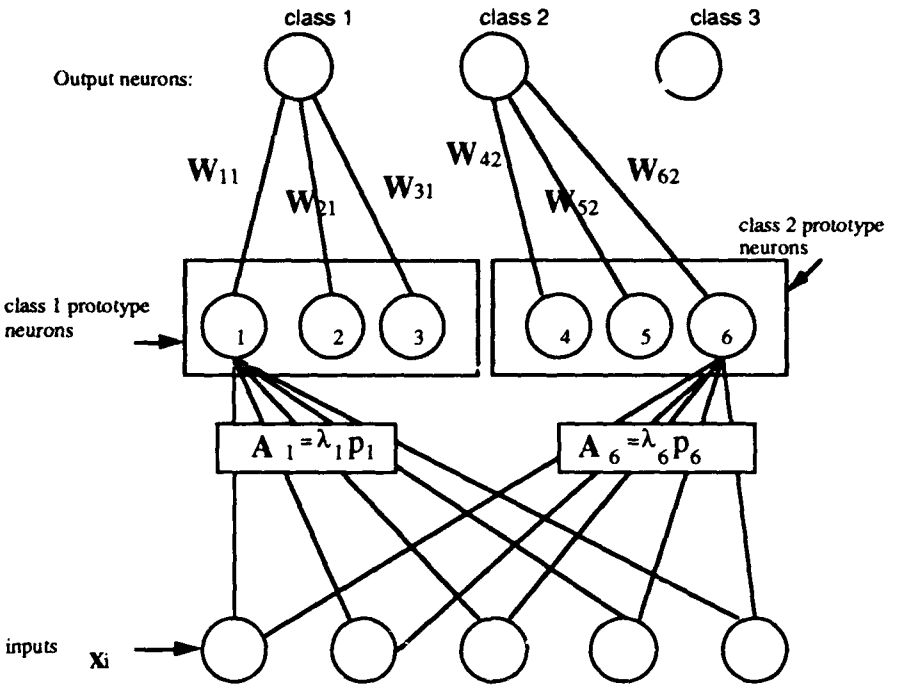


Fig. 3: RCE neural network architecture

The neural network operates in much the same way as the algorithm: Introduce the first input $\mathbf{x}_1$, and suppose it belongs to class 1. Create the first neuron of the second layer, corresponding to a prototype of class 1:

$$A_1 = \lambda_0 x_1 , \text{ and } W_{11} = 1 .$$

Now, introduce each input $\mathbf{x}_i$ vector and its class C_i=m, one at a time:

(1) If the correct class neuron (class m) gives a zero output, i.e., $h_m = 0$, then create a new second layer neuron (kth neuron) corresponding to prototype $\mathbf{p}_k$ of class m, i.e., set

$$A_k = \lambda_k x_i \qquad \text{with } \lambda_k = \lambda_0$$

and set W_{km}, the connection between neuron k of the second layer and the neuron corresponding to class m of the output layer equal to 1, i.e.,

$$W_{km} = 1.$$

(2) If an incorrect class output neuron, say class s, is activated, i.e.,

$$h_s =/ 0 \quad s =/ m,$$

then shrink the hypersphere radii of all active incorrect prototypes until the class s output neuron gives zero output, i.e., reduce λ_j until

$$h_s = 0.$$

Recall that $W_{km} = 1$ if the kth prototype is of the mth class, and $W_{km} = 0$ otherwise.

Testing can be performed on a set of inputs whose classes may or may not be known. When the input is presented, the output node that is activated (i.e., has a non-zero value) indicates the network's classification. If no output neurons or more than one output neuron is activated, then the input will not be classified correctly. The first situation can occur if some region of space was not covered during training and some new test input(s) is located in that region. The second case can occur if some input in the test set is located in a region of overlapping hyperspheres of different classes which were not adjusted during training since none of the training input vectors was in the overlapping region.

4. More General Hypersphere Classifiers

The more general hypersphere classifier allows moving and expanding in addition to shrinking the hypersphere in the learning process. Moving allows a nearby prototype vector of the correct (or incorrect) class to translate toward (or away from) a new input vector, much like the Kohonen learning procedure. Expansion allows a nearby hypersphere of the correct class to include a new input vector. In this way, prototypes do not necessarily coincide with an input vector and the hyperspheres can be adjusted to accommodate the data instead of creating new prototypes. The RCE network does not translate prototype vectors and it does not expand hyperspheres; only the creation of new prototypes and the shrinking of hyperspheres is allowed.

The disadvantage with the general hypersphere classifier is that translation or expansion of a hypersphere to include a new input vector in one region of space may exclude vectors which were properly classified inside the hypersphere and include inappropriate vectors inside the hypersphere. Such a scheme may prove unstable if the hyperspheres are alternately shifted one way then another or expanded then shrunk or a combination of both in order to accommodate input data in different regions. The RCE

classifier is generally more popular and is the basic unit of the Nestor Learning System (NLS).

5. Analogy to Coulomb Energy

Consider the Electrostatic problem in which a test particle of positive charge is placed in a space which includes some distribution of negative charges. Clearly, the positive charge will be drawn along some path into one of the negative charges. All of space can be classified according to which negative charge will be the eventual winner in attracting the test charge. The problem can be extended to more than 3 dimensions with a generalized Coulomb potential energy function

$$E = - \sum_i q_i / |x - \mu_i|.$$

The μ_i are the positions of the negative charges, and correspond to the prototypes; the q_i are the magnitudes of the charges and they determine the extent of the spatial region from which a positive test charge will be drawn to charge i.

If the problem is simplified by replacing the Coulomb potential with a square well potential, i.e.,

$$\begin{aligned} E_i &= 0 && |x - \mu_i| < \theta \\ &= -q_i/R_0 && |x - \mu_i| > \theta \end{aligned}$$

then we have what is unfortunately called a Restricted Coulomb Energy.

It is even more unfortunate that the hypersphere classifier is referred to as an RCE when only the creation and shrinking of hyperspheres is allowed. It seems that the name "Restricted Hypersphere Classifier" is more appropriate, leaving out the loose analogy to Coulomb Energy and Electrostatics.

6. Nestor NLS

The predominant use of the RCE network seems to be in the Nestor Learning Systems (NLS). The more sophisticated NLS combines a number of RCE networks in modular form. The method of their combination and details of the system are proprietary. Applications of NLS include signal classification (target recognition of sonar pings, heartbeat classification, and aircraft radar signature recognition), Image Processing (real-time 3-D object classification, industrial parts inspection, and signature verification), Character recognition (handwriting recognition and Japanese character recognition), speech recognition, and financial applications (automated mortgange insurance underwriting, mortgage delinquacy predection, automated securities trading, and bond trading).

7. Advantages and Disadvantages

7.1. Advantages

(1) The RCE network should learn very quickly since it uses only three layers, makes nodes only as it needs them, updates the weights in one step (shrinks the hypersphere immediately to exclude the inappropriate point) and has no weights to upgrade from the second to the third (output) layer. Also, not all the weights need to be updated when an input is introduced; only those which corrresp nd to an incorrectly activated prototype are altered. Furthermore, the updating of the weights to a single prototype is the same since only the multiplicative factor λ_j is reduced, i.e., $A_j = \lambda_j p_j$ ---> $A'_j = \lambda_j p'_j$.

(2) It is easy to map out the prototypes and their respective hyperspheres since each second layer node corresponds to one prototype and the weights connected to it from the first layer are related to the radii of the hyperspheres. Therefore, it is easy to visualize the separation of space by the hyperspheres of each class.

7.2. Disadvantages

(1) Typically, large numbers of prototypes are needed. Since each prototype coincides with some particular input vector, and cannot move, it is not placed optimally. For example, if the data inputs of one class fall in a simple sphere and the first input presented is off-center, then the first prototype will coincide with this input and it cannot include the whole sphere in its region of influence without including points outside the sphere. Since this prototype cannot move, more prototypes with smaller radii of influence will have to be added to include the regions left uncovered by the first prototype. Still, the entire sphere is not covered. This particular problem can be solved exactly with one prototype if it is moved to the center of the sphere and the hypershpere is expanded or shrunk until its radius is equal to that of the sphere.

(2) More input data may be needed to cover all of space with hyperspheres and remove ambiguous regions than typically required by back-propagation.

(3) Complicated boundaries between regions of different classes are not handled well by the RCE. A back-propagation network, gives a probabilistic value that a given point belongs to a particular class. As one approaches the boundary between two classes, the probabilities of the two classes may draw closer to each other reflecting the relative uncertainty of the network but also allowing a complicated boundary to be drawn by the network. The output of the RCE is digital; the boundary line is sharp. If it is incorrect, it can only be adjusted by including more input vectors, and new prototypes. Alternatively, the RCE network can be revised to include a probabilistic rather than a digital output in order to handle complex boundaries. Some of the modules in NLS may be RCE networks with probabilistic outputs.

(4) The RCE classifier may "overlearn" a fuzzy boundary. In other words, if there is some uncertainty or error in the input position, inputs of one class may be situated in the region of another. The RCE network will proceed to make isolated small hyperspheres around these inputs inside incorrect regions and shrink the correct class hyperspheres until they exclude these erroneous points. This problem can be diminished through "pruning", where hyperspheres which include just one (or very few) points are eliminated.

decay length	radial basis function	metric	correction method	PRCE	RCE+PRCE	Corrected PRCE	Corrected RCE+PRCE
0.1	Gaussian	Euclidean	1	8.2%	8.1%	7.4%	8.0%
0.5*RCE radius	Gaussian	Euclidean	1	7.4%	8.2%	8.5%	8.4%
0.5*RCE radius	Gaussian	Euclidean	2	7.4%	8.2%	7.9%	8.3%
0.1	Gaussian	City Block	1	7.8%	8.5%	7.2%	8.2%
0.1	Exponential	City Block	1	7.4%	8.1%	7.0%	8.1%

Table 1: RCE and PRCE performances on 2-dimensional separated Gaussians. Note that a) with variable deay lengths, the corrected weighs enhance the results, while with fixed lengths, the reverse is true, b) generally, RCE+PRCE is worse or no better than PRCE alnoe with or without weight corrections, and within the statistical error of 0.5%, and c) backpropagation is better at 6.7%.

decay length	radial bassis function	metric	correction method	PRCE	RCE+PRCE	corrected PRCE	corrected RCE+PRCE
0.2	Gaussian	City Block	1	5.3%	5.0%	5.4%	5.1%
0.1 * RCE radius	Gaussian	City Block	1	7.0%	7.0%	6.8%	6.8%
0.05 * RCE radius	Exponential	City Block	1	8.3%	7.9%	6.8%	6.7%
0.5	Exponential	City Block	1	4.9%	4.8%	5.3%	5.0%
0.5	Gaussian	Euclidean	1	1.7%	1.9%	1.9%	2.0%
0.2	Exponential	Euclidean	1	2.2%	2.3%	1.8%	2.1%

Table 2: RCE and PRCE performances on the 5-character recognition problem. Note that a) the Euclidean metric performs better than the city block metric, b) fixed decay lengths perform better than variable decay lengths with radial basis functions, and c) Differences in performance between PRCE, RCE+PRCE with and without corrected weights are within the statistical error of 0.5%.

8. References

1. Reilly, D.L., Cooper, L.N., Elbaum, C., "A Neural Model for Category Learning", Biological Cybernetics, v. 45, 1982, pgs. 35-41.

2. Scofield, C.L., Reilly, D.L., Elbaum, C., Cooper, L.N., Pattern Class Degeneracy in an Unrestricted Storage Density Memory, Nestor.

3. Reilly, et al., Learning System Architectures Composed of Multiple Learning Modules.

4. Lee, Y., Classifiers: Adaptive Modules in Pattern Recognition Systems, Masters Thesis, MIT.

5. Collins, E., Ghosh, S., Scofield, C., "Risk Analysis", DARPA Neural Network Study, AFCEA Intern. Press, Appendix G, pgs.429-443.

Radial Basis Approximations

Gregg Wilensky

1. Introduction

Given a set of prototypes, one can approximate the probability density function for class c as sum of radial basis functions $g_p(\vec{r}-\vec{r}_p)$ around each prototype p. The subscript p on the radial basis function indicates that the function may be different for each prototype. For example, the functions may be Gaussians with different standard deviations for each prototype. The problem at hand is to find the best set of coefficients w_{cp} to weight each of the basis functions in order to best approximate the class probability density $p(\vec{r}|c)$:

$$p(\vec{r}|c) \cong \sum_{p'} w_{cp'}\, g_{p'}(\vec{r}-\vec{r}_{p'}).$$

I will show two approaches. But first we must recognize that the probability density is normalized to unity:

$$\int (d\vec{r})\, p(\vec{r}|c) = 1,$$

which implies

$$\sum_{p'} w_{cp'} = 1,$$

since the radial basis functions are assumed to be normalized:

$$\int (d\vec{r}')\, g(\vec{r}-\vec{r}') = 1.$$

The first method of approximating the coefficients involves recognition of another approximation, the Parzen's windows approximation, in which the probability density for class c is approximated by a sum over all the data points of radial basis functions around each data point:

$$p(\vec{r}|c) \cong \sum_{a} p_c^{(a)}\, g(\vec{r}-\vec{r}^{(a)}),$$

where the coefficients are defined by

$$p_c^{(a)} = \frac{\bar{p}_c^{(a)}}{\sum_{a'} \bar{p}_c^{(a')}}, \qquad \bar{p}_c^{(a)} = \begin{cases} 1: & a \text{ not of class } c \\ 0: & a \text{ is of class } c \end{cases}.$$

In order to estimate the coefficients for the prototype expansion we will minimize the integrated squared difference of the probability densities when estimated by these two approaches:

$$\varepsilon \cong \int (d\vec{\mathbf{r}}) \sum_{c'} \left[\sum_{p'} w_{c'p'} g_{p'}(\vec{\mathbf{r}} - \vec{\mathbf{r}}_{p'}) - \sum_{a} p_c^{(a)}\, g(\vec{\mathbf{r}} - \vec{\mathbf{r}}^{(a)}) \right]^2.$$

The derivative of this error with respect to each weight will be zero when the error is minimized. This derivative can be written as:

$$\frac{\partial \varepsilon}{\partial w_{cp}} = \sum_{p'} M_{pp'} w_{cp'} - v_{cp},$$

where

$$M_{pp'} = M_{p'p} = \int (d\vec{\mathbf{r}}) g_p(\vec{\mathbf{r}} - \vec{\mathbf{r}}_p) g_{p'}(\vec{\mathbf{r}} - \vec{\mathbf{r}}_{p'})$$

and

$$v_{cp} = \sum_{a} p_c^{(a)} \int (d\vec{\mathbf{r}}) g_p(\vec{\mathbf{r}} - \vec{\mathbf{r}}_p) g(\vec{\mathbf{r}} - \vec{\mathbf{r}}^{(a)}).$$

One can either solve for the weights by inverting the matrix:

$$w_{cp} = \sum_{p'} M^{-1}_{pp'} v_{cp'}$$

or gradient descent can be used to iteratively approach the solution:

$$\delta w_{cp} \propto -\frac{\partial \varepsilon}{\partial w_{cp}} = -\sum_{p'} M_{pp'} w_{cp'} + v_{cp}.$$

In the special case in which the radial basis functions are Gaussian and the metric is Euclidean:

$$g_p(\vec{r}-\vec{r}_p)=\frac{1}{\left(2\pi\sigma_p^2\right)^{N/2}}\exp\left[-\left(\vec{r}-\vec{r}_p\right)^2\Big/2\sigma_p^2\right],$$

$$g_p(\vec{r}-\vec{r}_p)=\frac{1}{\left(2\pi\sigma_p^2\right)^{N/2}}\exp\left[-\left(\vec{r}-\vec{r}_p\right)^2\Big/2\sigma_p^2\right],$$

the matrix elements take on a simple form which is just another Gaussian:

$$M_{pp'}=g_{p+p'}(\vec{r}_p-\vec{r}_{p'})=\frac{1}{\left(2\pi(\sigma_p^2+\sigma_{p'}^2)\right)^{N/2}}\exp\left[-\left(\vec{r}_p-\vec{r}_{p'}\right)^2\Big/2(\sigma_p^2+\sigma_{p'}^2)\right]$$

$$v_{cp}=\frac{1}{\left(2\pi(\sigma_p^2+\sigma^2)\right)^{N/2}}\sum_a p_c^{(a)}\exp\left[-\left(\vec{r}^{(a)}-\vec{r}_p\right)^2\Big/2(\sigma_p^2+\sigma^2)\right].$$

If all prototypes have the same standard deviation then the diagonal elements of M are all constant and the off-diagonal elements are smaller than the diagonal elements. One can approximate the inverse of a matrix $(1+A)^{-1} \approx 1 - A + A^2$ ... Keeping only the first two terms in the expansion gives the estimate for the weights:

$$w_{cp}\cong\sum_a p_c^{(a)}\exp\left[-\frac{\left(\vec{r}_p-\vec{r}^{(a)}\right)^2}{4\sigma^2}\right]-\sum_{p'\neq p}\exp\left[-\frac{\left(\vec{r}_p-\vec{r}_{p'}\right)^2}{4\sigma^2}\right]\sum_a p_c^{(a)}\exp\left[-\frac{\left(\vec{r}_{p'}-\vec{r}^{(a)}\right)^2}{4\sigma^2}\right]$$

If instead of using Gaussian radial basis functions with a Euclidean metric we use exponential basis functions with a city block metric,

$$|\vec{r}-\vec{r}'|=\sum_i|\vec{r}-\vec{r}'|_i,$$

$$g_p(\vec{r}-\vec{r}')=\frac{\kappa_p}{2}\exp\left[-\kappa_p|\vec{r}_p-\vec{r}_{p'}|\right],$$

then the matrix has a more complicated form:

$$M_{pp'}=\prod_i M_i,$$

where

$$M_i = \frac{1}{2}\frac{\kappa_p \kappa_{p'}}{\kappa_p^2 - \kappa_{p'}^2}\left\{\kappa_p \exp\left[-\kappa_{p'}\left|\vec{r}_p - \vec{r}_{p'}\right|_i\right] - \kappa_{p'} \exp\left[-\kappa_p\left|\vec{r}_p - \vec{r}_{p'}\right|_i\right]\right\}.$$

For the case in which $\kappa_p = \kappa_{p'}$, this reduces to:

$$M_{pp'} = \frac{\kappa}{4}\exp\left[-\kappa\left|\vec{r}_p - \vec{r}_{p'}\right|\right]\prod_i\left(1 + \kappa\left|\vec{r}_p - \vec{r}_{p'}\right|_i\right).$$

Lynch-Granger Model of Olfactory Cortex

1. Introduction

Richard Granger and Gary Lynch at the University of California, Irvine have developed several unsupervised neural network models. Their approach is to model as faithfully as possible a biological network (the olfactory system) and to extract the essential features of the model. Higher and higher level models, which simplify the details while capturing the function of the biological models, are then constructed. This approach has led them to models of learning and memory in the olfactory system. These are hierarchical clustering models which group a set of input vectors (representing various odors, for example) into clusters of similar cues.

Though the model could be applied to unsupervised clustering with any types of inputs, I will use the example of smell to illustrate it. An odor is input to the primary olfactory sensors, which input to the lateral olfactory bulb. The bulb may normalize the inputs and send them to the olfactory cortex. The pattern of activity in cortex encodes information about the odor. The patterns resulting from the response to the first sniff of an odor will be similar for similar odors (Note that rats sniff at a rate of about 5 Hz). For example, all sweet smelling odors may give the same (or very similar) first sniff response in the cortex. The second sniff to an odor classifies the odor at a different hierarchical level. For example, the first sniff response to flowers and fruit may be the same because they are both sweet odors. A second sniff may give the same response to any type of flowery smell (roses or lilies, for example) but will give a different response to a fruity smell. The third sniff may discriminate between various types of flowers or various types of fruits.

In the simplest algorithmic form of the model (discussed in the preprint "*Simulation of paleocortex performs hierarchical clustering*", to be published in *Science*), the hierarchical clustering is achieved by combining a winner-takes-all network with feedback to modify the input signal (smell). A weighted sum of inputs from the bulb (1st layer) feeds into the cortical nodes (2nd layer). This weighted sum is a vector inner product ----- $\mathbf{w}_1 \cdot \mathbf{x}$ between the input vector $\mathbf{x}$ and the weight vector $\mathbf{w}_1$ associated with node number 1. There is a separate weight vector for each node in the 2nd layer. The node with the largest inner product (the largest overlap with the input vector) is selected to win the competition. This winning set of weights is then trained so that it moves in the direction of the input vector: $\mathbf{w} \rightarrow \mathbf{w} + \gamma(\mathbf{x} - \mathbf{w})$. When trained on many odors, similar odors will light up the same node and the above training law will result in $\mathbf{w}$ approximating the mean of the associated input vectors. The next step, after a node wins and its weight vector is modified is to present a modified input vector to a second network (representing the second level in a hierarchical tree). The input vector is modified by subtracting from it the winning weight vector: $\mathbf{x} \rightarrow \mathbf{x} - \mathbf{w}$. This is

equivalent to translating the origin in the input vector space to the mean of the cluster (the winning weight vector). The above procedure is repeated for this second network, whose winning weight vectors will learn the mean of the subclusters. This process is repeated for all the networks. The result is that successive networks learn to cluster the inputs with finer grades of distinction.

The above model is not perfect. Similar subclusters can interfere with each other; subcluster weight w_{12} (which should represent the mean of points in subcluster 1 within main cluster 1) may learn the mean of points clustered about both main cluster 1 and main cluster 2. This is overcome in the more complete olfactory model which incorporates collateral connections within the cortex. In other words, there is feed-forward excitation (or the equivalent effect from initiation of a refractory period in the inhibitory connections) which selectively excites neurons associated with the winning node. In this way, the neurons which learn a subcluster in cluster 1 will be different from the neurons which learn a subcluster in cluster 2. A second effect of the feed-forward connections (the effect emphasized in the preprint "*Asymmetry in cortical networks enhances hierarchical clustering*") is the sharpening up of the clustering. It tends to make the response to similar cues more similar and the response to different cues more different.

The more detailed and biologically faithful model includes both the olfactory bulb and the olfactory cortex. Feedback from the olfactory cortex to the bulb is responsible for modifying the inputs after each sniff so that the replacement $\mathbf{x} \rightarrow \mathbf{x} - \mathbf{w}$ is mimicked. Feed-forward collateral connections within the cortex both sharpen up the classification and remove possible confusion among subclasses, as mentioned above. The connections from the bulb to the cortex (along the lateral olfactory tract, LOT) feed the normalized olfactory signals emerging from the bulb to the cortex. These connections are random and sparse. This sparseness allows subsequent sniffs to sample different parts of the input which allow finer levels of distinction. The Hebb-like learning rule, based on long-term potentiation (LTP), only allows synaptic strengths (weights) to increase.

Within both the bulb and the cortex are patches of neurons with local inhibitory connections that can effect a winner-takes-all response. Modification of the number of winners in each patch results in different types of clustering analyses (more on this in the more complete report).

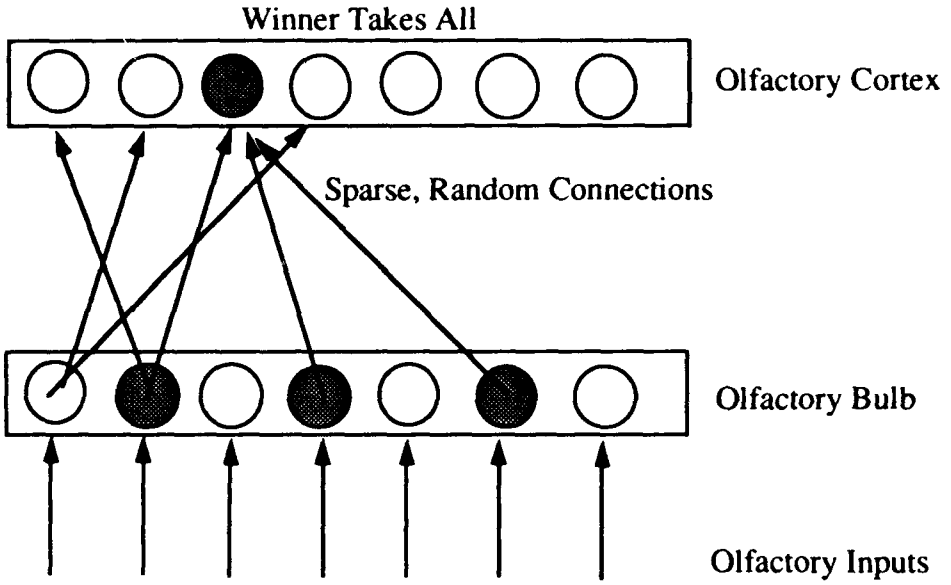


Fig. 1: Hierarchical clustering model of Lynch and Granger

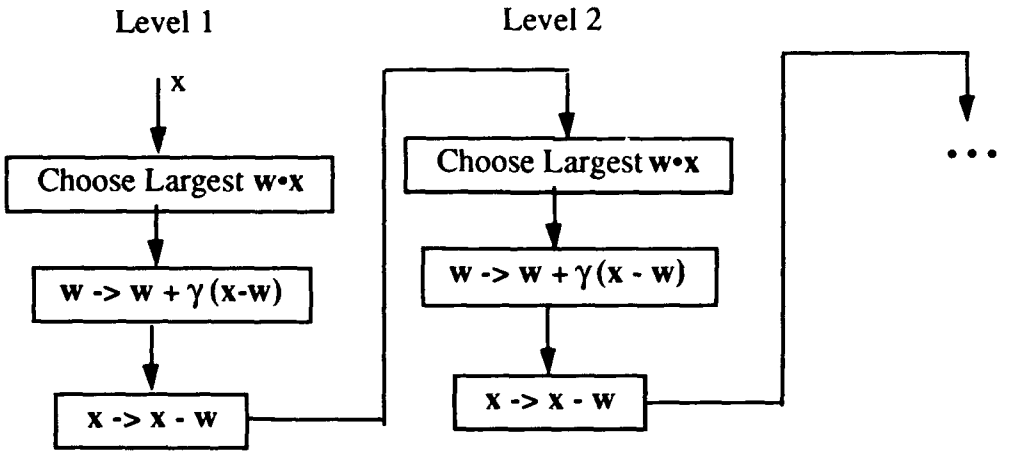


Fig. 2: Simplified hierarchical clustering model of Ingerson, Granger and Lynch

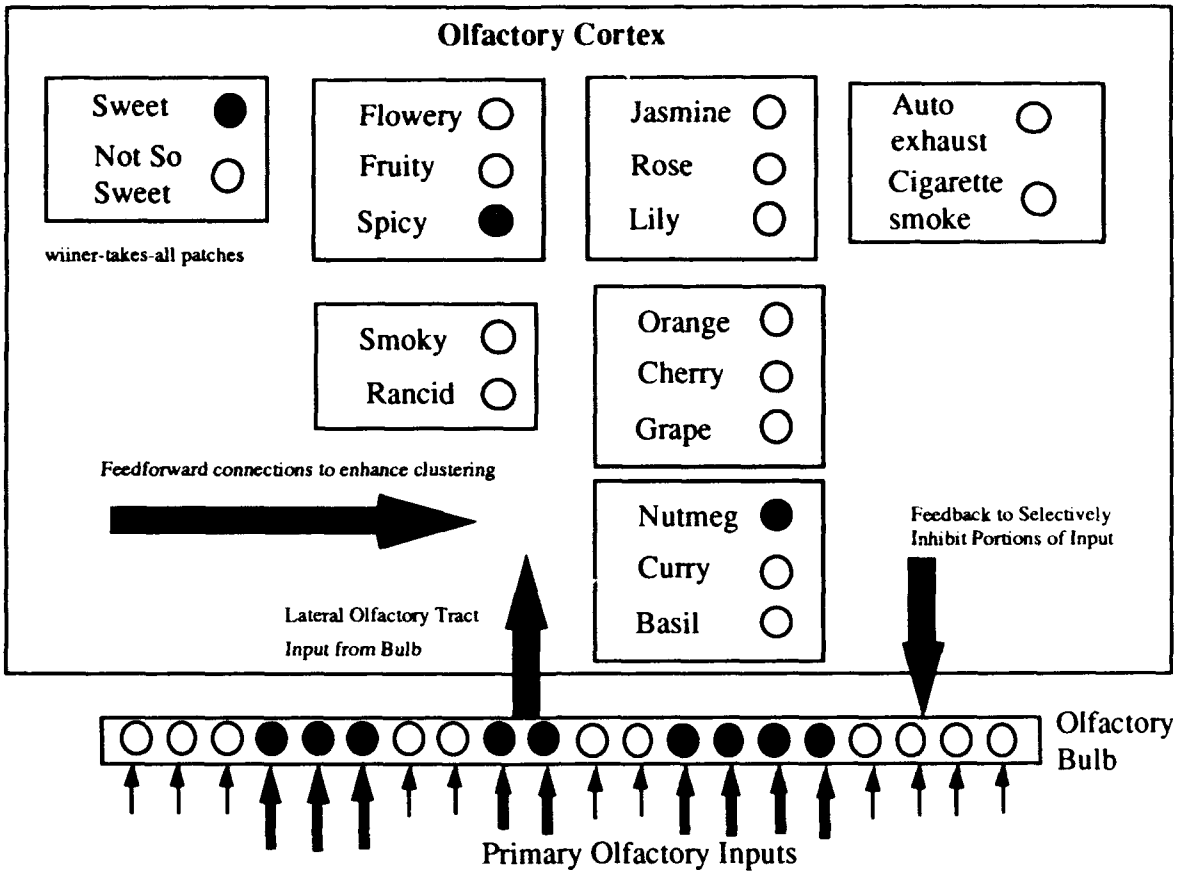


Fig. 3: Illustration of olfactory model.

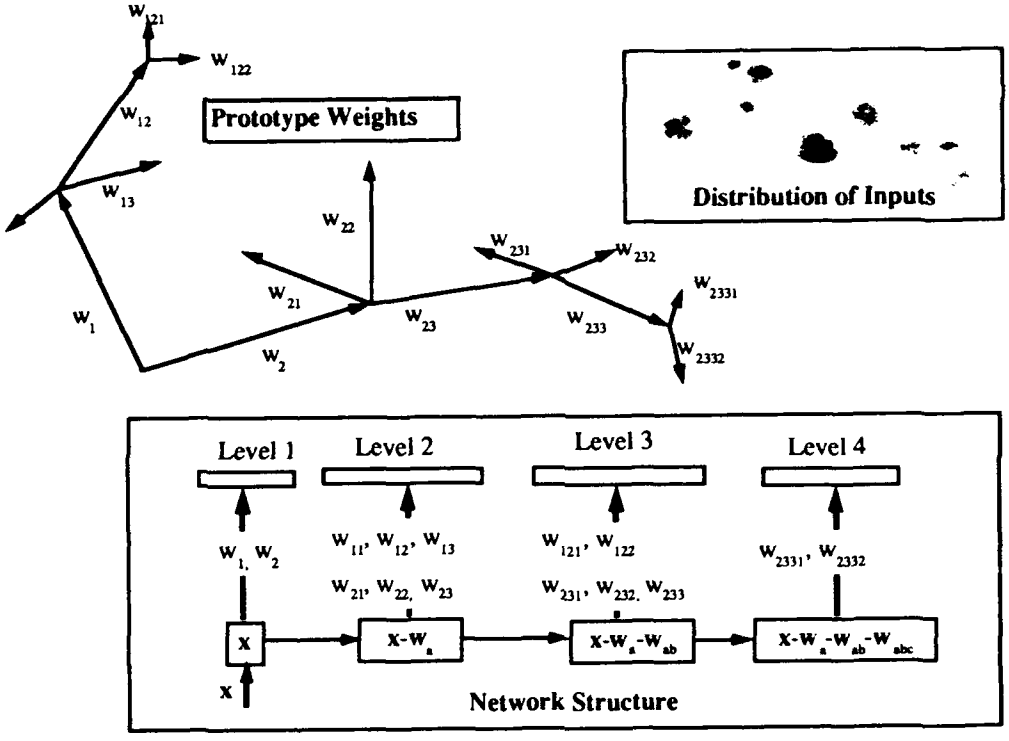


Fig. 4: Hierarchical layers of prototypes

Multi-parameter Process Control with Neural Networks

Narbik Manukian, Joseph Neuhaus, Gregg Wilensky[19]

November 1992

Abstract

Neural networks (NN's) are used to predict characteristics of GaAlAs layers grown by organometallic chemical vapor deposition (OMCVD). Because of the scarcity of training data available for this high dimensional problem, a procedure, the successive approximation method, was developed which enables the most relevant input parameters to be selected first by a linear NN and then used by a more general NN. In addition, by training to predict the correction to physical formulae for the layer characteristics, maximum use is made of prior knowledge about the problem. Indeed, this procedure results in a significant improvement in predictive capability beyond the simple physical formulae.

1. Introduction

The control of a set of process parameters by many input parameters is a problem of interest in many real industrial applications. The manufacture of products often involves a process controlled by several parameters, where the product must meet quality requirements or performance specifications. Sometimes a basic physical formula is used to partially or approximately describe the output as a function of the inputs and then predict the appropriate input values for a desired output. However, the actual output deviates from the simple theoretical prediction, and a more refined or accurate description is required. This study focuses on the application of neural networks to control the growth of GaAlAs solar cells[20] by OMCVD. The growth process is controlled by the setting of approximately 30 parameters, and the outputs investigated in this study are the thicknesses and dopant concentrations of specific layers of the solar cell. As a first approximation, the thickness and dopant concentration can be expressed in terms of several inputs analytically as will be discussed later. However, these predictions are not sufficiently accurate, and a more precise mapping of the inputs to the outputs is desired. In this study, the neural network model is used to improve the analytic approximation in determining the thicknesses and dopant concentrations of solar cell layers. However, the general techniques developed here are also applicable if no analytic or empirical formulae exist. Any technique that attempts to map the inputs to the outputs in such process control problems must address the characteristic problems that occur in real manufacturing operations.

---

[19]This research was sponsored by DARPA/ONR contract N00014-89C-0257.

[20] The data for this study was obtained from Kopin Corporation.

First, the parameters must be properly scaled, since they are in different units with different ranges of variation. In the solar cell growth process, the parameters consist of various pressures, temperatures, flows, etc., some of which have little or no variation (0 - 5%), others have a limited set of discrete values (2 or 3), and some have large variations, where the largest value of the parameter is several times the smallest value, and still others such as the dopant concentrations vary by several orders of magnitude. Several normalizations of the parameters are possible, but care must be taken so that the normalization does not obscure or unduly emphasize a parameter or its variation. A small value or variation in an input may cause that particular input to be ignored, while a very large value or variation in an input may saturate a neuron on the next layer. One advantage of a neural network is that it can readily mix parameters in different units successfully, since the weights can also serve as unit conversions.

With sparse data, any technique that attempts to correlate outputs to a large set of inputs or approximates outputs as functions of the inputs must guard against 1) correlations among inputs and 2) random correlations between inputs and outputs. Correlations among inputs can result from physical relationships among the inputs and make a unique solution impossible. For example, in a dilute gas, the temperature, pressure and volume of a gas are related, and since any one of the related parameters can be considered as dependent on the other two, there can be no unique solution for the relationship of these inputs to the outputs. Thus, any particular solution may over- or under-estimate the importance of a dependent parameter. Correlation between inputs or outputs can also result from a decision of the experimenter to run the process only in desirable regimes of the output space. For example, in the solar cell growth experiments, thickness and dopant concentration are intentionally correlated as shown in Fig. 1, so that thin layers tend to be n-doped and thick layers tend to be undoped. Without prior knowledge that these correlations are imposed by the experimenter, any technique, including a network, will assume that the correlations are real.

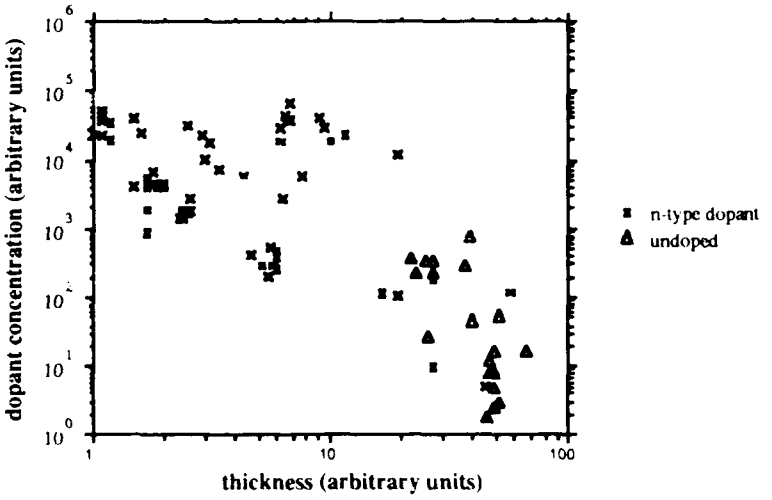


Figure 1. Experimenter-induced correlation between thickness and dopant concentration[21].

[21] Note: Units have not been provided for the data in this report in order to protect the confidentiality of Kopin Corporation's data.

Random correlations between inputs and outputs are the most prevalent problem that can occur with a small training sample and a large input dimension. As an example, consider a set of irrelevant input parameters which are known to have no effect on a particular output parameter. There is always some finite probability that the changes in the output parameter will be correlated to the random changes of one of these inputs or a combination of these inputs, simply because some of the small fluctuations of these parameters just happen to match the changes in the output. Then, given a sufficient number of input parameters and a sufficiently sparse data set, such random correlations are likely to occur.

In the solar cell growth experiment, the problem with random input-output correlations prevented a straightforward application of a back-propagation network using all the input parameters as input nodes. A typical back-propagation network with a particular random initial setting of weights would key in on some mixture of significant and insignificant parameters while another network starting from a different random initialization of the weights would key in on a different mixture of input parameters. There is no a-priori way of ensuring a unique and physically meaningful result from any particular run of a simple back-propagation network. Furthermore, there is not enough data to train a network with such a large input dimension and determine the proper output without overtraining.

Principal component analyses are also ineffective here, since these analyses emphasize the role of the input parameters which lead to the greatest output variation, rather than those input parameters which are most effective in predicting the output. The result is that the first few principal components contain many of the irrelevant or unimportant parameters mixed with the important parameters in some combination, rather than just the linear combination of important parameters.

The following sections describe an approach to overcome these problems through the successive neural net approximation. The method is then applied to the solar cell growth problem with positive results.

2. The Neural Network Model

2.1. The Leave-One-Out Training Method

Neural networks generally require large amounts of training data to generalize well on difficult problems. Since the Kopin solar cell data is sparse (on the order of 100 points) we need to guard against the possibility of over-fitting the training data. This is done by partitioning the data into two sets; one is used for training the NN, and a distinct set is used to test the generalization performance. One must trade off the amount of data preserved for training against the remaining available for testing. In order to maximize the amount of training data, we have implemented the leave-one-out method in which only one data point is preserved for testing. The single data point is extracted from the training set and used to test the performance of a network which has been trained on all

remaining points. This procedure is repeated until all points in the data set have been tested. If there are N points in the data set then we require N networks, each one is trained on N-1 points and produces test results for a single point. It is important to make sure that each network is trained starting from randomly initialized weights. This technique is computationally intensive but minimizes statistical error in the results and provides a means of quantifying the generalization performance.

2.2. Neural Network Outputs: Successive Approximations Using Back Propagation

To solve the problem of process control with neural networks where the outputs are controlled by a large number of input parameters and the training data set is limited, we need to develop a technique which reduces or eliminates the effects of random correlations and can be used to improve upon an existing analytic or empirical approximation. Our approach is to calculate successively finer approximations of the output by iterating through smaller networks with more manageable input sets. Two back-propagation networks are used in each iteration as illustrated in Fig. 2.

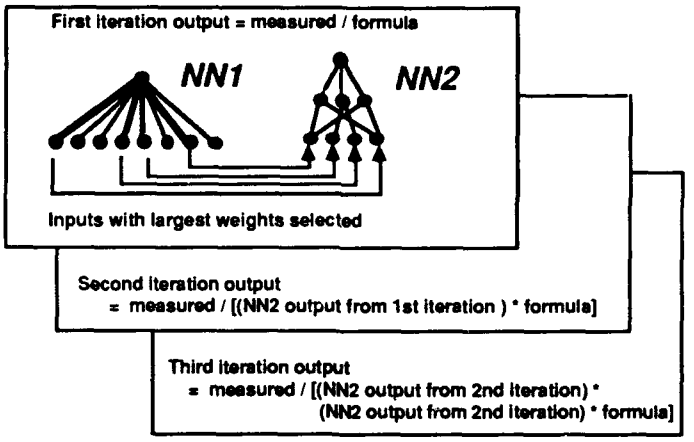


Figure 2. Successive approximation technique: Repeated iterations with two neural networks.

The first network (NN1) is designed to reduce the input set by selecting the parameters in order from greatest to least correlated to the output. NN1 performs only a linear correlation so that it determines separately for each parameter how strongly it is correlated to the output. Using the leave-one-out method, a separate NN1 is trained on each training set leaving out a different data point. For each NN1, the absolute value of the weights to each input parameter determines the relative importance of that parameter. These absolute values of the weights to each input parameter
are then averaged over all training sessions, and these averages determine which parameters have the greatest impact on the input-to-output mapping. By averaging the absolute values of the weights over all NN1s, the problem with undesirable correlations is reduced: If there are dependent parameters, then different networks will favor one dependent parameter over another but the average weights of these parameters will have a more uniform emphasis. Averaging over all NN1s also reduces the effect of more

complicated random correlations between inputs and outputs, since these correlations are unlikely to recur in every network. Simple correlations, e.g. between a single parameter and the output, are less likely to occur and they may not be removable through any technique without additional data points. Some variables which may be selected as significant by NN1 can be rejected based on physical arguments, and others because they have little or no variation over the entire data set.

The second network (NN2) in each iteration trains on a reduced, and therefore more manageable, set of input parameters obtained from NN1. It is not a linear correlator, but a standard 3 layer back-propagation network which can approximate non-linear output functions. The smaller number of input parameters greatly reduces the probability of overlearning due to random correlations between inputs and outputs. In a more exhaustive study, the optimum number of input parameters could be determined empirically by trying to solve the problem with only the parameter with the largest average weight, and then repeatedly adding inputs in order of decreasing average weights, until the network shows no improvement. However, in this study, we chose the first few parameters with average weights distinctly larger than the rest of the parameters. If through this selection, an important parameter is omitted, then it may be included in the next iteration. If an unimportant parameter is included as a result of some random correlations, then it will probably be ignored by NN2.

If there exists an a-priori analytic or empirical approximation (formula), then the first iteration of the networks NN1 and NN2 are trained to output

$$\text{Desired Output} = \frac{\text{Measured Value}}{\text{Formula}}$$

as shown in Fig. 2, rather than training directly on the measured value. If there are no formulae, then the first iterations will be trained to output the measured value directly. Training on this ratio allows the first iteration to learn corrections of the formula rather than attempting to determine the functional dependence of the output on the inputs directly. Then, each successive iteration is trained to output the measured value divided by the approximation of the output in the previous iteration of
NN2 as shown in Fig. 2. Thus, each successive iteration of the network will minimize the fractional error relative to the previous approximation. The iteration scheme stops when NN1 cannot separate any parameter as more significant, or when NN2 can no longer improve on the last approximation.

In this study, we used existing analytical formulae that approximate the thickness and dopant concentration shown below in Table 1. The variable names are capitalized. Those beginning with the letter F are gas flows, those beginning with P are pressures and those beginning with T are temperatures. In addition to these variables, t is the event time, X is an additional process parameter, and RUN is the experiment number used as a rough indication of time. A detailed description of the parameters is not provided in this report in order to protect the confidentiality of Kopin Corporation's process.

Quantity	Analytical Formulae
n-Type Dopant Concentration	$\frac{(F6)(F8)}{F6+F11}\Big/\left(\frac{F1}{P1}e^{-E_1/k(T_1+273)}+2\frac{F2}{P2}e^{-E_2/k(T_2+273)}\right)$
Layer Thickness	$t\times\left(\frac{F1}{P1}e^{-E_1/k(T_1+273)}+2\frac{F2}{P2}e^{-E_2/k(T_2+273)}\right)$

Table 1. Analytic formulae for the outputs used as a first approximation.

2.3. Neural Network Inputs: Normalizing Input Parameters

The Kopin data consists of measurements of various parameters during an event of wafer layer growth. One or more events may contribute to the growth of a single layer on the wafer. During an event, parameters are set to a constant value or ramped linearly to achieve a desired effect. For each event, the average, minimum, maximum and standard deviation of each parameter are recorded. For this study, we only used the average values of the parameters over an event.

To address the problem of combining these parameters which have different physical units and different ranges of variation, we employ the statistical characteristics of the data to scale the inputs to the neural network. The manner in which data is normalized and presented to the network, especially a back-propagation (BP) network, greatly influences the network's ability to find a robust solution. It is also preferable to normalize the input parameters in a way that matches the dynamic range of the hidden layer neuron activation. For each parameter x, we find the mean and standard deviation over the entire data set of N events:

$$\bar{x}=\frac{1}{N}\sum_{j=1}^{N}x_j \quad \text{(mean)}$$

$$\sigma=\sqrt{\frac{1}{N-1}\sum_{j=1}^{N}(x_j-\bar{x})^2} \quad \text{(standard deviation)}$$

Then, we scale each parameter with respect to its mean and standard deviation over the data set before feeding it to the neural network:

$$x_{input}=\frac{x_{measured}-\bar{x}}{\sigma} \quad \text{(NN input).}$$

This normalization has the advantage that each parameter will vary in the same range about zero, but its disadvantage is that small variations in parameters which only have small variations will be magnified. In such cases, unphysical or irrelevant correlations between such variations and the output must be avoided.

There are a total of 142 data points in the raw Kopin data file. Depending on the type of prediction being made, a subset of these points is used for training and testing purposes. For example, if the network is attempting to learn n-type doping then all undoped type data points are excluded from the training set. Below is a table of the number of points used for each case.

Type of Prediction	# Points Used	# Parameters Used
Layer Thickness	130	28
n-Type Dopant Conc.	67	31

Table 2. Summary of available Kopin Data.

3. Results

3.1. Layer Thickness predictions

As discussed before, the first set of networks (NN1) is designed to determine the best linear fit of the inputs to the outputs. The average of the absolute value of the weights from each input to the output (thickness) is shown in Fig. 3 for prediction of single layer thickness. The input parameters F_1, F_3, t, F_2, F_4, and X, an additional process parameter, were chosen as the most significant parameters for the first approximation, and fed as inputs to the second network. P_1 was omitted because its set value was constant for the entire data set and its measured value varied by less than 0.16%. Note that all parameters, even the irrelevant parameters were considered important by some networks, demonstrating the necessity of averaging over many networks and only choosing a few parameters whose average weights are distinctly higher than the rest of the parameters.

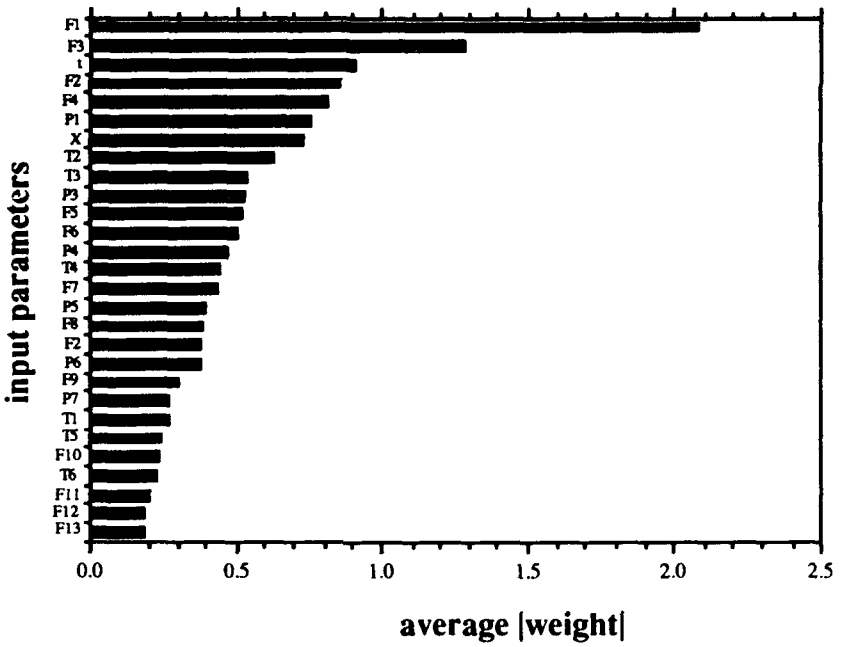


Figure 3. Average weights from linear networks (NN1): correlation of inputs to layer thickness.

With these 6 parameters as inputs, NN2, was run with varying numbers of hidden nodes in order to determine the optimum number of hidden nodes for generalization. The resulting performance on the training and test sets with the leave-one-out method using 1,3,5,10 and 20 hidden layer nodes shows that the test results are relatively insensitive to the number of hidden nodes (see Fig. 4). We chose to run the network with 10 hidden layer nodes since the results were slightly better with 10 nodes.

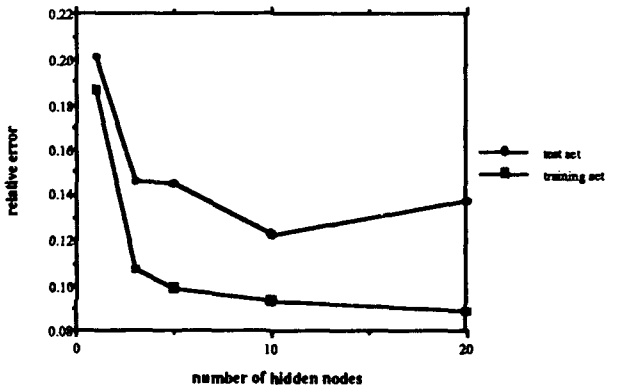


Figure 4. NN2 generalization capability as a function of the number of hidden layer nodes in predicting layer thickness.

Finally, the second network, shown in Fig.5, was trained with these 6 parameters as inputs, 10 hidden layer nodes, using the leave-one-out method to output the desired thickness divided by the analytical approximation. The improvement of the network output over the formula is shown in Fig. 6, where the error made by the network is generally less than the error made by the formula. The run number indicated is a reference number provided by Kopin for each experiment. The overall improvement by the network in estimating the thickness is shown in Table 3: The average relative error made by the network is 12.3% as compared to 27.5% made by the analytical formula averaged over the entire data set. This corresponds to an average reduction of the relative error by a factor of 2. The limited data set did not allow another neural network iteration in the successive approximation scheme. The next iteration of the linear approximation network (NN1) did not select any parameters as significant above the rest of the parameters. A greater amount of data is needed to further improve the prediction.

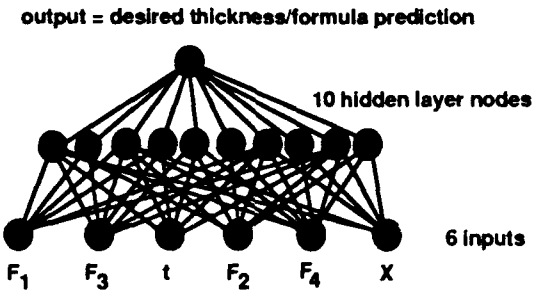


Figure 5. Neural network (NN2) used for approximation of layer thicknesses.

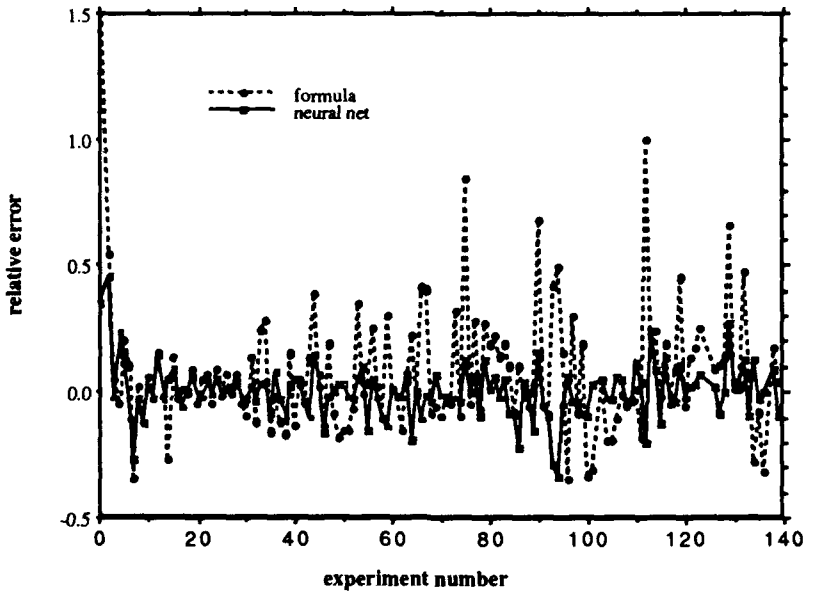


Figure 6. Improvement over the analytic approximation by the neural network model in predicting layer thickness.

	Layer Thickness	n-type Dopant Concentration
Analytic Approximation	27.5%	46.9%
Neural Network Approximation	12.3%	19.9%

Table 3. Relative Error of Formula vs. Network.

3.2. Dopant Concentration Prediction

For n-type dopants, the linear correlation of inputs to outputs analyzed by NN1 resulted in 3 parameters with average weights noticeably above the other parameters as shown in Fig. 7. The run number, RUN, is incorporated here in order to account for possible degradation of dopant concentrations over time.

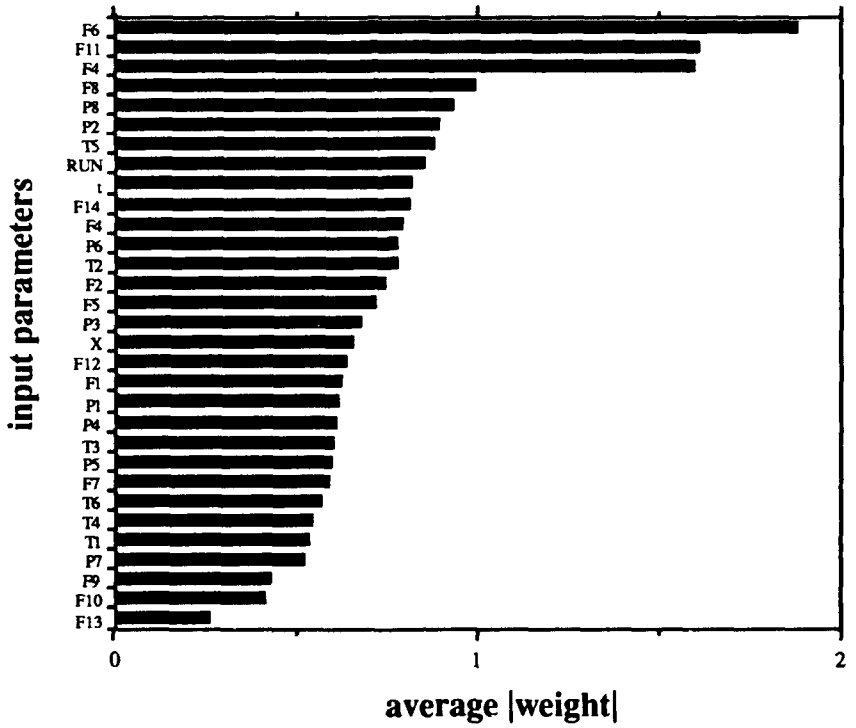


Figure 7. Average weights from linear networks (NN1): correlation of inputs to n-type dopant concentrations in the first iteration.

After NN2 is trained on these parameters in the first iteration of the successive approximation technique, a set of NN1 networks are then applied in the second iteration of the successive approximation scheme. They are trained to output the desired dopant divided by the thickness as determined by NN2 in the first approximation. This procedure yielded 4 parameters with average weights somewhat greater than other parameters as shown in Fig. 8. Of these 4 parameters, only F_1 had a set value that varied over the data set. The other 3 parameters were rejected. Since training the second iteration of NN1 selected out just this single parameter as an important input, we chose to add F_1 to the first iteration of NN2 and retrain it on the data set rather than training the second iteration of NN2 on a single parameter.

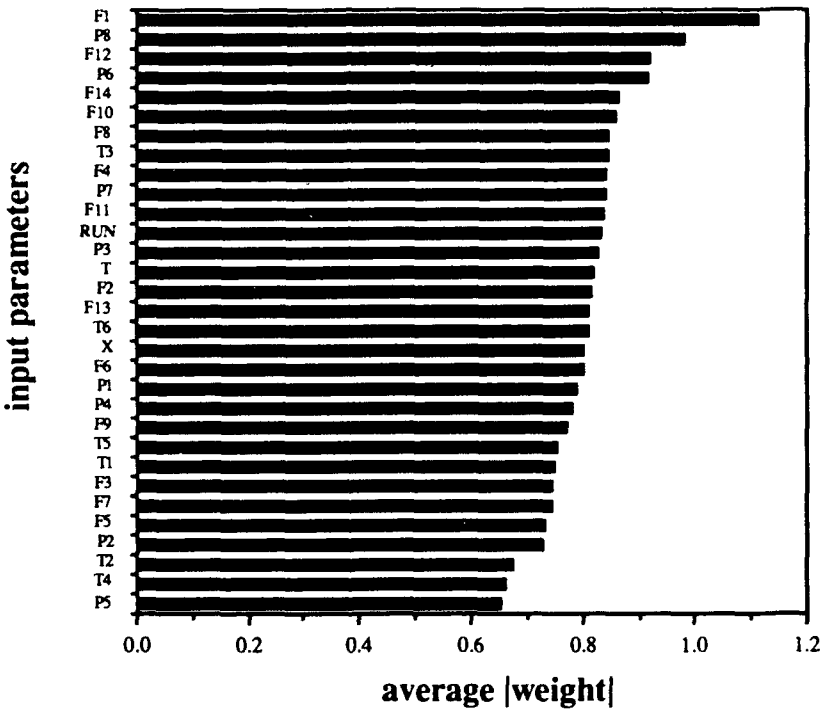


Figure 8. Average weights from linear networks (NN1): Correlations of inputs to n-type dopant concentrations in the second iteration.

Thus, NN2 was trained with four inputs, F_1, F_3, F_6, F_{11}, with varying numbers of hidden nodes as shown in Fig. 9, where each point represents the average performance of independently trained NN2 networks. The training error is relatively independent of the number of hidden layer neurons. We chose 3 nodes in the hidden layer and trained each network to output the n-type dopant concentration divided by the analytic approximation (see Fig. 10) using the leave-one-out method. The performance is shown in Fig. 11, where the network once again improves on the analytic approximation. The average improvement is again approximately a factor of 2: The relative error made by the

network in the n-type dopant concentration is 19.9% as compared to 46.9% by the formula as shown in Table 3. Also, the long time performance of a single NN2 network was tested to make certain that the networks were stopped at an appropriate point in their training and that further training would add little improvement. As shown in Fig. 12, the network was stopped at 40,000 trials and further training did not substantially enhance its performance.

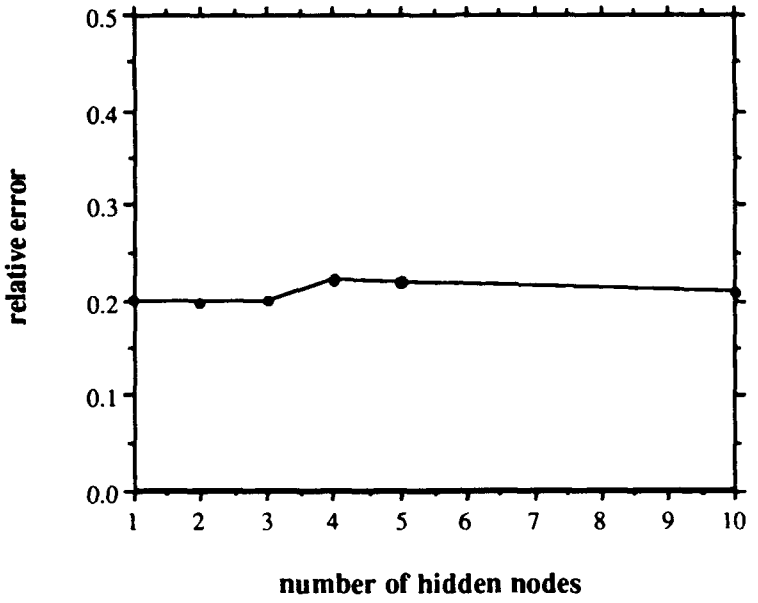


Figure 9. Generalization capability as a function of the number of hidden nodes in predicting n-type dopant concentrations.

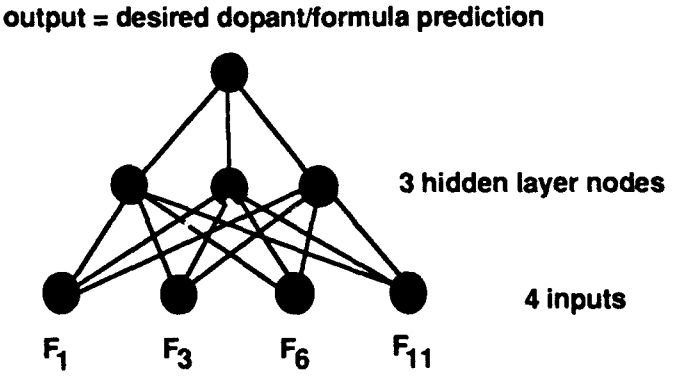


Figure 10. Neural network used for approximation of n-type dopant concentrations.

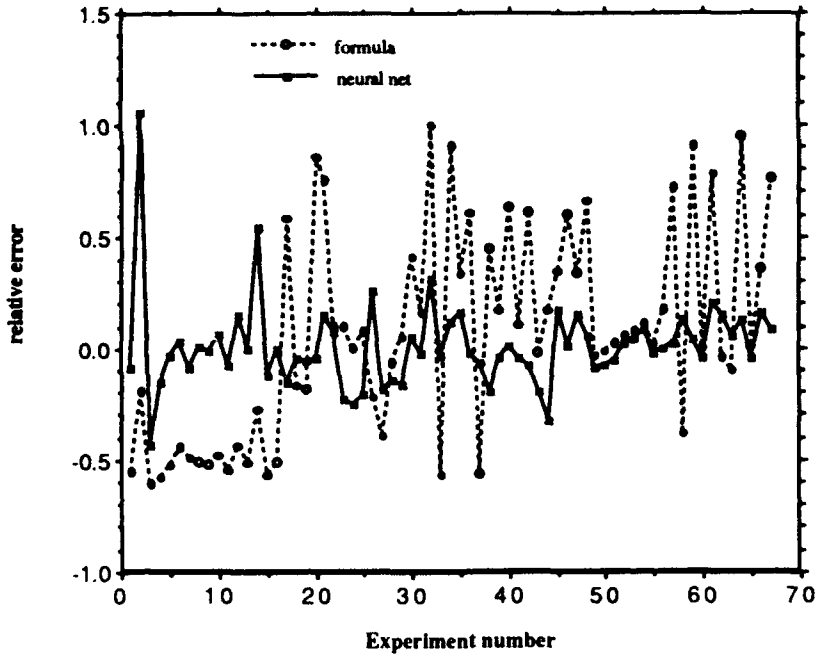


Figure 11. Improvement over the analytic approximation by the neural network model in predicting n-type dopant concentrations.

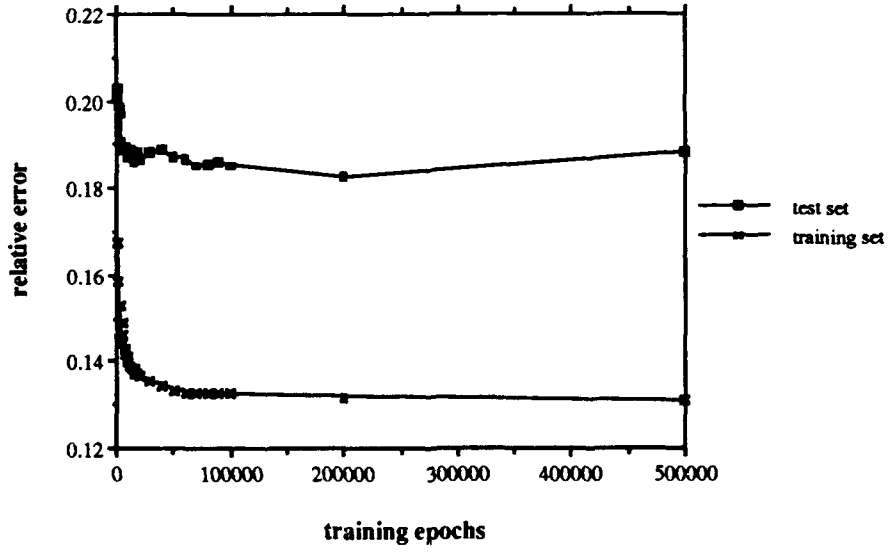


Figure 12. Performance of NN2 after longer training on n-type dopant concentrations.

4. Conclusions

We have developed a neural network model and a successive approximation technique for industrial process control in order to predict the desired outputs of a process as a function of the inputs. Specifically, we have applied it to the growth of layers in a solar cell where the number of input parameters which can potentially control the process is large, such that the data points cannot sufficiently describe the dependence of the output on the inputs. We have also addressed the problem of random correlations between inputs and outputs and designed our model to avoid such correlations whenever possible. We used two neural networks: The first network selected important parameters through a linear correlation and the second network used this reduced set of inputs to estimate the output. Further, the model is designed to improve upon an existing empirical or analytical formula, and we have demonstrated the improvement over the formulae in approximating both the thicknesses and the dopant concentrations of the solar cell layers by at least a factor of 2. Finally, the model is designed to successively iterate through both networks and obtain progressively finer approximations of the output.

5. Acknowledgments

We would like to thank Matthew Zavracky and Ronald Gale of Kopin Corporation for providing the data for this study and for their assistance in understanding and interpreting it. We are also grateful to Natalie Rivetti for her assistance in editing and compiling this report.

Detection of Ocean Wakes in Synthetic Aperture Radar Images with Neural Networks[22]

Gregg Wilensky, Narbik Manukian, Joe Neuhaus, John Kirkwood
Logicon/RDA

Abstract

Two neural networks are combined to detect wakes in Synthetic Aperture Radar (SAR) images of the ocean: The first network detects local wake features in smaller subportions of the image, and the second network integrates the information from the first network to determine the presence or absence of a wake in the entire image. The networks train directly using the gradient descent method on either real SAR images or on synthetic images and are designed to detect wakes in images with low signal-to-noise ratios. When trained on real images, the network detector recognizes the wake in any translation and is robust with respect to rotations. With synthetic images, the network model is able to recognize wakes with all possible translations, rotations and over a wide range of opening angles. The performance of the neural network is measured as a function of the signal-to-noise ratio in synthetic images and as a function of a parameter related to the signal-to-noise ratio in real images. The network outperforms the human eye in detecting wakes in both real and synthetic images.

1. Introduction

Detection of low signal-to-noise synthetic aperture radar (SAR) images of wakes on the ocean surface is a difficult problem, and existing template matching techniques suffer from large computational expense and the difficulty of developing realistic templates. Neural networks have been very successful in solving pattern recognition and classification problems particularly when the rules for classification are either unknown or difficult to specify. In image recognition, neural networks have proven to be robust with respect to the degradation of the signal-to-noise ratio and distortions of the image such as translations, scaling, and rotations. In this paper we demonstrate the construction and performance of a neural network designed to detect ship wakes in low signal-to-noise SAR images of the ocean surface. We train and test the network model on both real SAR images and synthetic images generated to provide a larger statistical sampling as well as greater control over the signal-to-noise ratio.

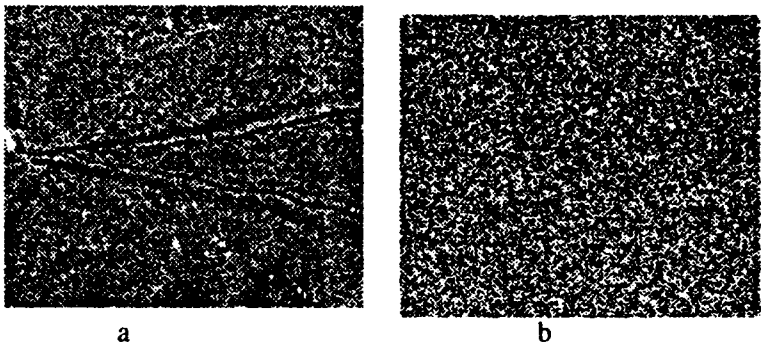
---

[22]This research was supported by Logicon RDA internal research and development funding (9001-0004), and the extensions to compare with human visual capabilities were sponsored by DARPA/ONR contract N00014-89C-0257. This paper also appears in Government Microcircuit Applications Digest of Papers, vol. 18, Nov. 1992.

2. The Images

2.1. Real SAR images and the addition of noise

The real SAR images are produced from a single digital image of a wake obtained from experiments at Loch Linnhe, Scotland. From this single image, we construct smaller 512 x 512 sections of the image such that about half the images contain the wake and the other half contain only noise. An example of a wake containing section is shown in Fig. 1a. The starting pixels of the sections are chosen randomly so that the wake images may contain the wake in any position (translation) within the image.



a b

Figure 1. a) Example of a wake image without added noise obtained from original image b) Example of a wake image with added noise at a noise level of 3.0.

The distribution of pixel intensities in one-look SAR images in the pure noise and the wake regions of the image is approximately exponential:

$$p_n(I_i) = (1/I_n)\exp(-I/I_n) : \text{noise}$$

$$p_s(I_i) = (1/I_{si})\exp(-I/I_{si}) : \text{wake (signal)} \qquad (1)$$

With such distributions, it is difficult to add noise and reduce the signal-to-noise ratio without disturbing the exponential form of either distribution. But, it is possible to add noise to the entire image and preserve the exponential distribution of the noise sections which comprise the majority of the image. Since the wake distribution is close to that of the noise, its exponential character will be altered only slightly.

To do so, let the distribution of the additional noise ΔI be

$$p(\Delta I) = (I_n / I_n')\,\delta(\Delta I) + (1 - I_n / I_n')(1 / I_n')\exp(-\Delta I / I_n'),$$

where I_n is the average pixel intensity in the noise region of the image and $\delta(\Delta I)$ is the Dirac delta function. Then the final distribution will preserve the exponential form, since at $\Delta I \neq 0$, the first term is zero and only the exponential term survives, and when $\Delta I = 0$, no noise is added. However, the additional noise is always greater than or equal to zero,

and thus the overall average intensity of the noise pixels is increased so that <I'> is greater than <I>. By adding to each pixel the difference between the initial and final average pixel intensities, $(I_n - I_n')$, the average intensity of the pixels will be preserved, i.e.

$$<I''> = <I'> + (I_n - I_n') = I_n = <I>.$$

The "noise level" is defined by the ratio

$$r = I_n' / I_n$$

and its relationship to the signal-to-noise ratio is shown in the Appendix. An example of an image containing a wake with added noise of noise level of 3.0 which is near or beyond human detection capabilities is shown in Fig. 1b.

2.2. Synthetic wakes and the signal-to-noise ratio

The synthetic images allow us to train and test the network on a statistically significant number of images with a controlled signal-to-noise ratio and all possible translations, rotations and opening angles of the wake. The synthetic images are 256x256 pixel images where each pixel intensity in a noise or wake region is generated from the exponential noise and signal distributions found in real images so that the resulting distribution of pixel intensities is similar to that of real images (1). A smaller image size of 256x256 is chosen in order to speed up the training time, and each wake image is constructed with random variations in the position, and orientation of the wake and with opening angles varying randomly from 7 to 28 degrees. With a known distribution of intensities for both signal and noise pixels, the signal-to-noise ratio can be determined in terms of the ratio of average signal to average noise pixel intensities I_{si}/I_n as derived in the Appendix. Therefore, by choosing an average noise pixel intensity I_n for all images, any desired signal-to-noise ratio can be produced by an appropriate choice of the local wake pixel intensity I_{si}.

3. The Model

A single back-propagation neural network wake detection model with the entire SAR image as its input vector is time consuming to train and does not have built-in translational or rotational invariance. Instead, we have constructed a two-neural network wake detector as illustrated in Figure 2. Both networks are 3 layer (1 hidden layer) back-propagation networks trained with gradient descent. The first neural network (nn1) processes smaller 32 x 32 pixel "templates" of the image and detects the presence of some portion of a wake arm within the template. Its outputs are the probability that some part of the wake arm is present within the template and the most probable angle of the wake with respect to the horizontal. Since the templates can be chosen from any part of the image, nn1 is capable of translationally invariant detection of local features. Dividing the entire image into templates and operating nn1 on each template effectively transforms the entire image into a "reduced template image", where all the pixel values

within a template are replaced by the outputs of the first network. The second neural network (nn2) takes the reduced template image output by nn1 as input and determines whether or not a wake is present in the entire image. It integrates the information from nn1, i.e. the presence of the wake and its orientation within each template is correlated with that of other templates.

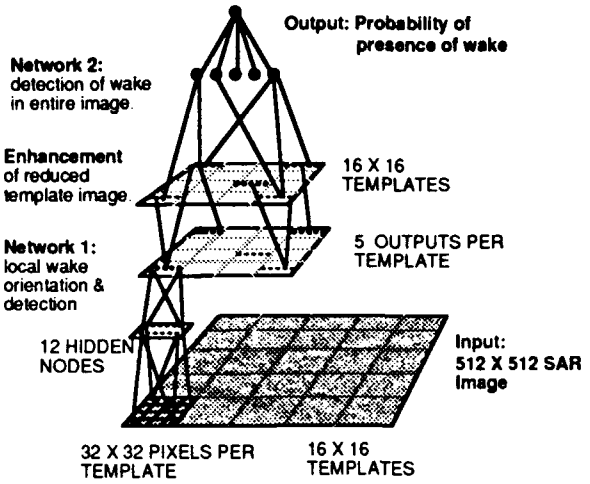


Figure 2: Two neural network model operating on a SAR image

3.1. The first neural network

As illustrated in Fig. 2, nn1 is trained to detect local wake features (a portion of a wake arm) in a template. The intensity of each pixel is fed directly as input into the first layer of nn1. Since local features of the wake consist mainly of a band or bands of pixels with intensities which are either higher or lower than the background noise, nn1 must be trained to recognize a light or dark band of pixels with an appropriate distribution of pixel intensities in many positions and orientations within the template. The weights to the intermediate layer nodes can be initially set to be sensitive to such bands at various angles and positions within the template. This initialization saves much computation time, otherwise, with a random initial setting of the weights, the training time of nn1 will be unreasonably long.

When training on real images, nn1 has one intermediate layer of 12 nodes whose weights are initially set to detect bands which are 8 pixels wide, oriented in 4 different directions, 0, $\pi/4$, $\pi/2$, and $3\pi/4$ and in three different positions for each orientation, as shown in Fig. 3. After initialization, the weights and thresholds of all the nodes are allowed to vary during training in order to reach a minimum of the error in the outputs. However, the essential band structure of the intermediate layer weights for both the synthetic and real image problems is not lost, although there are some changes which include the curving of the bands, the creation of double banded templates, and the appearance of considerable heterogeneity in the pixel intensity across the templates. The picture of some of the weights after training on synthetic images is shown in Fig. 4 and

illustrates that the band character of the weight templates is maintained through the training process.

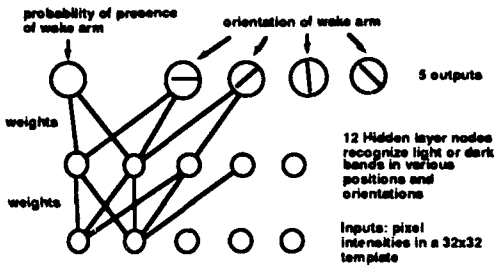


Figure 3. Feature detection neural network (nn1).

hidden node 1 2 3

Figure 4. Weights to some hidden layer nodes after training.

As illustrated in Fig. 2 and 3, nn1 processes each template as a separate input and reduces it to 5 outputs per template. The first output corresponds to the probability that the template includes the wake as determined by nn1. If the template includes any portion of the wake, the desired output of the first output node (n=1) is 1.0, and if the template includes only pure noise, it is 0.0. The next four outputs, n=2-5, are designed to give the direction of the wake in the template. For each template, the angle of the wake, with respect to the horizontal within the template, θ, is determined, where $0< \theta <\pi$. The maximum values of the 4 angular outputs are equal to 1.0 and occur when $\theta = \theta_n$, where θ_n is 0, $\pi/4$, $\pi/2$, and $3\pi/4$, for n = 2, 3, 4, and 5, respectively. Given θ, the desired output p_n for each angular node is

$$p_n = (1 - |\theta - \theta_n| / d\theta) \quad \text{if } |\theta - \theta_n| < d\theta ,$$

$$p_n = 0 \qquad \text{otherwise,}$$

where $d\theta$ is the angle between successive node maxima, i.e. $\pi/4$. Note that when θ is between θ_n and θ_{n+1}, the desired values p_n and p_{n+1} of the angular output nodes n and n+1 will be between 0 and 1 according to how close θ is to θ_n and θ_{n+1}. All other angular nodes will have a desired value of zero.

When training on real images, the templates are chosen at random, starting from any pixel in the image, and care is taken to avoid starting points which would result in templates extending beyond the image boundary. This random template choosing process maximizes the variety of templates, and forces nn1 to recognize a portion of the wake with varying widths in any position and orientation within the template. When training on synthetic images, a new noise or wake template is constructed at each training step, with a random position and orientation of the wake for wake-containing templates.

When training on real or synthetic images, nn1 is trained with equal frequency on wake and noise templates.

3.2. The second neural network

The second neural network (nn2) is trained on the outputs from nn1 for an entire real or synthetic image. With real images, a different image is chosen for each training trial. With synthetic images, a new image is generated for each trial. Then, each real or synthetic image is fed into nn1 in the testing mode. The templates are chosen in order from the upper left to the lower right covering the entire section with no gaps, and they are fed into nn1 one at a time. The combined output of nn1 for all the templates is then fed into nn2. There is just one output for nn2 and its desired value is 1 if the image includes a wake, and it is 0 if it only consists of noise.

3.3. Image enhancement techniques

Image enhancement methods applied to the original image or to the "reduced template" image can improve the performance of the neural network and provide a clearer picture of the wake for human observers. An iterative template enhancement technique can be used to improve images of wake arms in the reduced template image. As shown in Figure 2, the reduced template image produced by nn1 can be enhanced through an iterative scheme which updates each template according to the state of its neighboring templates in the following manner: Let p be he probability that template i,j contains part of the wake as predicted by nn1. Of the eight nearest neighboring templates of i,j, examine only the two neighbors which are closest to the y direction with respect to i,j. Let p_+ and p_- be the probabilities of the presence of a wake in these two neighboring templates. Then, enhance the probability of the presence of a wake in i,j to p', where

$$p' = p + p\,(1 - p)\,(\,|p_+ - p| + |p_- - p|\,)/2\,,$$

The second term on the right is the enhancement of p, and is designed to have stable points at p=0, p=1, and $p_+=p_-=p$. Thus, successive enhancements of the probability will tend to converge to either extreme, 0 or 1, or when all three neighbors have equal probabilities.

4. Results

4.1. Performance on real images with added noise

For each iteration in the training of nn1, a new template is chosen randomly from a real image with a noise level r = 2.0. Then, nn2 is trained and tested cn wake and noise sections of the real image with variable noise levels, r=1.5, 2.0, 2.5, 3.0, 3.5, and 4.0. In a more thorough study nn1 can be trained on a range of noise levels.

The fraction of wake and noise images missed as a function of the noise level r is shown in Fig. 5. The neural network model outperforms the average performance of 5 persons at every noise level. Note that the neural network detector did not miss any wake or noise images when tested at or below a noise level of 1.5. At a noise level of 3.0 (example in Fig. 1b), the neural network correctly identifies between 80% and 85% of the wake and noise images, and it performs well below this level with 75% correct at a noise level of 3.5, and between 60% and 70% correct at a noise level of 4.0 as shown in Fig. 5.

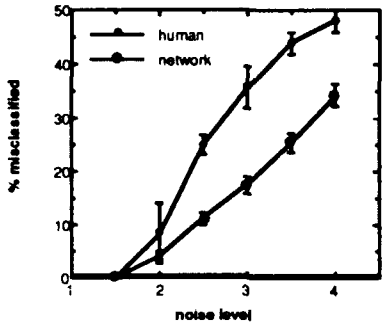


Figure 5: Detection capability of the neural network model vs. the human eye as a function of the noise level on real SAR images.

The single output of nn2 corresponds to the probability that the image section being tested contains a wake. However, we are free to choose the cutoff probability p_C according to our emphasis, and trade between the fraction of the noise and wake sections misclassified. The misclassification of wake and noise sections as a function of p_C at a noise level of 3.0 is shown in Fig. 6, where the tradeoff between false alarms and missed wakes is evident as p_C is varied from 0 to 1.

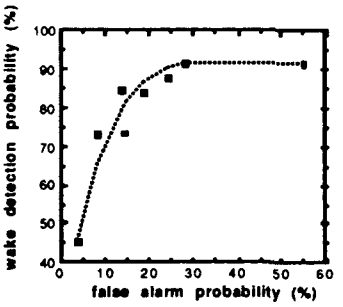


Figure 6: Performance of the network as a function of the cutoff p_C on real images at a noise level of 3.0.

4.2. Performance on synthetic images

With synthetic images, a new 256 x 256 wake or noise image is constructed for each iteration of the training process. Since the synthetic images contain wakes in any

position and orientation and a range of opening angles, a neural network trained on these images is capable of translationally and rotationally invariant wake detection and must be robust with respect to opening angles of the wake. nn1 is trained on images at one signal-to-noise ratio (s/n = 20 dB), but nn2 is trained separately on a range of signal-to-noise ratios, namely s/n = 15, 20, 25, and 30 dB.

Its performance is compared to the average performance of 5 persons on the same images and is shown in Fig. 7. The neural network outperforms humans at the lower signal-to-noise ratios, 15 and 20 dB, but not at the higher signal-to-noise ratios, 25 and 30 dB. This results from training nn1 only on one signal-to-noise ratio, and it can be improved by training nn1 separately at each signal-to-noise level. However, in this study, the emphasis was on demonstrating the network's detection capability at signal-to-noise ratios below that of human detection capabilities.

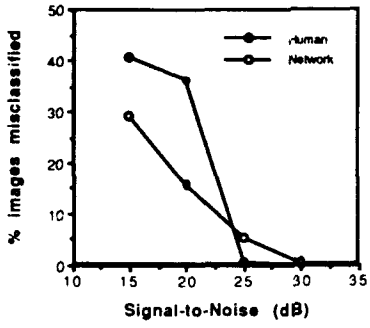


Figure 7: Detection capability of the neural network model and the human eye on synthetic images as a function of signal-to-noise ratio.

5. Conclusions

A neural network model for automated detection of wakes on the ocean surface in SAR images has been developed. It can detect wakes in both real and synthetic images which are beyond the detection capabilities of human vision. It has also proven to be insensitive to rotations and translations of the wake and robust with respect to a range of opening angles of the wake. The performance of the network has been measured as a function of the signal-to-noise ratio in synthetic images and as a function of the noise level which is related to the signal-to-noise ratio in real images. The tradeoff between false negatives and false positives has been measured as a function of the threshold which can be set to any desired value.

6. Appendix:

The Signal-to-Noise Ratio in an Image with Exponential Distribution of Pixel Intensities

The pixel intensities in noisy SAR images are found to follow an exponential distribution. In a "one-look" image, the probability that a particular pixel i will have an intensity I_i is

$$p_s(I_i)=\frac{1}{I_{si}}e^{-I_i/I_{si}} \quad , \quad p_n(I_i)=\frac{1}{I_n}e^{-I_i/I_n}$$

for images which contain a signal or only noise, respectively. I_{si} is the local average intensity in the neighborhood of the ith site in an image-containing portion, and I_n is the average pixel intensity over the noise only part of an image. For a "four-look" image,

$$p_n(I_i)=\frac{4^4 I_i^3}{3!\, I_n^4}e^{-I_i/I_n} \quad , \quad p_s(I_i)=\frac{4^4 I_i^3}{3!\, I_{si}^4}e^{-I_i/I_{si}}.$$

All the above probabilities are normalized so that their integral over I_i from 0 to ∞ is 1. The probability that a particular set of intensities $\{I_i\}$ will occur is the product of the single site probabilities over the entire image:

$$p_n\{I_i\}=\Pi_i p_n(I_i) \qquad p_s\{I_i\}=\Pi_i p_s(I_i).$$

Since the SAR image used in this paper is a four-look image, we shall evaluate the signal-to noise ratio for a four-look image. The average intensities over an ensemble of noise and signal images are I_n and I_{si} which can be verified by integrating Ii over the distributions. The average of I_iI_j over an ensemble of noise images is

$$\langle I_iI_j\rangle_n=\int_0^\infty \Pi_k dI_k I_iI_j p_n\{I_k\}.$$

For i≠j,
$$\langle I_iI_j\rangle_n=\int_0^\infty dI_i I_i p_n(I_i)\int_0^\infty dI_j I_j p_n(I_j)=I_n^2.$$

For i=j,
$$\langle I_iI_j\rangle_n=\int_0^\infty dI_i I_i^2 p_n(I_i)=\frac{5}{4}I_n^2.$$

The standard deviation of the pixel intensities in pure noise images is

$$\langle I_iI_j\rangle_n=\langle I_i\rangle_n\langle I_j\rangle_n=\frac{1}{4}I_n^2\delta_{ij}.$$

The signal-to-noise ratio is defined in terms of χ, the log of the ratio of the signal to noise probability distributions:

$$\chi = \log \frac{p_s\{I_i\}}{p_n\{I_i\}} = 4\sum_i \left[\left(\frac{1}{I_n} - \frac{1}{I_{si}} \right) I_i - \log \frac{I_{si}}{I_n} \right].$$

The ensemble average of χ over the noise images and the signal images is

$$\langle \chi \rangle_{n,s} = 4\sum_i \left[\left(\frac{1}{I_n} - \frac{1}{I_{si}} \right) \langle I_i \rangle_{n,s} - \log \frac{I_{si}}{I_n} \right]$$

and the standard deviation of χ over the noise images is

$$\sigma_n^2 = \langle \chi^2 \rangle_n - \langle \chi \rangle_n^2 = 16\sum_{i,j} \left[\left(\frac{1}{I_n} - \frac{1}{I_{si}} \right) \left(\frac{1}{I_n} - \frac{1}{I_{sj}} \right) \langle I_i I_j \rangle_n - \langle I_i \rangle_n \langle I_j \rangle_n \right]$$

$$= 4\sum_{i,j} \left(\frac{1}{I_n} - \frac{1}{I_{si}} \right)^2 I_n^2 .$$

The signal-to-noise ratio s/n is defined as

$$s/n = \frac{\langle \chi \rangle_s - \langle \chi \rangle_n}{\sigma_n} = \frac{2\sum_i \left(\frac{I_{si}}{I_n} + \frac{I_n}{I_{si}} - 2 \right)}{\left(\sum_i \left(\frac{1}{I_n} - \frac{1}{I_{si}} \right)^2 I_n^2 \right)^{1/2}} .$$

which is identical except for the factor of 2 to the expression of s/n for one-look images.

The Projection Neural Network[23]

Gregg D. Wilensky and Narbik Manukian

Logicon RDA
6053 W. Century Blvd.
Los Angles, California 90045

Abstract

We develop a new neural network model, the projection neural network, which overcomes three key drawbacks of backpropagation-trained neural networks (BPNN): 1) long training times, 2) the large number of nodes required to form closed regions for classification of high dimensional problems, and 3) the lack of modularity. This network combines advantages of hypersphere classifiers, such as the restricted Coulomb energy (RCE) network, radial basis function methods, and BPNN. It provides the ability to initialize nodes to serve either as hyperplane separators or as spherical prototypes (radial basis functions) followed by a modified gradient descent error minimization training of the network weights and thresholds which adjusts the prototype positions and sizes and may convert closed prototype decision boundaries to open boundaries and vice versa. The network can provide orders of magnitude decrease in the required training time over BPNN and a reduction in the number of required nodes. We describe the theory and give example applications. A U.S. patent on this Projection Neural Network is pending.

1. Introduction

Neural networks (NN) have been applied with success to a wide range of pattern classification and function fitting problems. The standard backpropagation training algorithm[24], while successful for problems of moderate size, suffers from slow training times, the potential to get stuck at local error minima, and the need for a large number of nodes when applied to complicated problems. However, in problems for which it does converge to a solution, it offers the advantage of ensuring error minimization. As a consequence, when solving a classification problem, the network outputs will approach the Bayes conditional probabilities, given a statistically representative set of training data. On the other hand, there exist classification algorithms which train quickly but do not guarantee minimization of the classification error. Examples of these are the hypersphere classifiers, such as the reduced Coulomb energy network (RCE)[25], the models of

---

[23]The development and theoretical parts of this research were supported by Logicon RDA IR&D funding, and the applications were supported by DARPA/ONR contract N00014-8?C-0257. A U.S. patent for the Logicon Projection Network™ has been allowed and is pending. This paper has been published in the International Joint Conference on Neural Networks, Volume II, 1992, pp 358-367.

[24]Rumelhart, D. E., & McClelland, J. L. Parallel Distributed Processing: Explorations in the Microstructure of Cognition (Vol. 1,2). Cambridge, MA: MIT Press., 1986.

[25] Reilly, D. L., Cooper, L. N., Elbaum, C. "A Neural Model for Category Learning". Biological Cybernetics, v. 45, 1982, pgs. 35-41.

adaptive resonance theory (ART)[26,27], and the Kohonen type networks[28]. In this paper we present a neural network which combines the utility of both approaches.

The classification algorithms which provide fast training do so by placing prototypes with closed decision boundaries around training data points and then adjusting their positions and/or sizes. As an example, a hypersphere classifier such as RCE places hyperspherical prototypes around training data points and adjusts their radii. Radial basis function networks can provide fast training as well as error minimization[29,30,31,32]. While several methods of determining the size, position and amplitude of the radial basis functions have been proposed they do not have the simplicity or computational efficiency of backpropagation training[33]. In contrast, the projection network provides a means of implementing radial basis functions with a uniform approach to learning these parameters: backpropagation training of the weights and thresholds of a feedforward network. This effectively leads to optimization of the prototypes' locations, sizes and amplitudes. Furthermore, both closed decision regions (hyperspheres or hyperellipses) and open ones (such as hyperplanes) are accommodated in the same network. Training of the network parameters may convert closed decision regions to open ones and vice versa in the process of minimizing the error.

Feedforward neural networks such as BPNN with at least one hidden layer of sigmoidal nonlinearities are capable of approximating any mapping of a continuous bounded function of N inputs into N' outputs[34]. They accomplish this by partitioning the input space with hyperplanes whose positions and orientations are determined by the weights and the thresholds of the hidden layer nodes. The nonlinear combination of such hyperplanes can lead to partitioning of the input space into regions bounded by hyperplanes and curved surfaces, both closed and open. To form a closed region in N dimensions requires at least $N+1$ hyperplanes. The more complex the classification boundaries, the more regions will be needed (although the number is reduced somewhat by sharing hyperplanes among regions). For large N this can lead to an excessively large network. In contrast, the projection network can combine closed prototypes with open prototypes , and therefore requires only one node per closed region.

---

[26]Carpenter, G. A. & Grossberg, S. "ART 2: Self-organization of Stable Category Recognition Codes for Analog Input Pattern". Applied Optics v. 26, 1987, pgs. 4919-4930.

[27]Carpenter, G., Grossberg, S., Rosen, D. "ART 2-A: an adaptive resonance algorithm for rapid category learning and recognition". IJCNN 1991 II, 1991, pgs. 151-156.

[28]Kohonen, T. (1986). "Learning Vector Quantization for Pattern Recognition", *Technical Report TKK-F-A601*, Helsinki University of Technology.

[29]Broomhead, D. S. & Lowe, D. "Multivariable Functional Interpolation and Adaptive Networks". Complex Systems, v. 2, 1988, pgs. 321-323.

[30]Powell, M. J. D. "Radial Basis Functions for Multivariable Interpolation: A Review.". J. C. Mason and M. G. Cox (Eds.), Algorithms for Approximation Clarendon Press, Oxford. 1987, pgs. 143-167.

[31]Moody, J. & Darken, C. "Fast Learning in Networks of Locally Tuned Processing Units". Neural Computation v. 1(2), 1989, pgs. 281-294.

[32]Poggio, T. & Girosi, F. "Networks for approximation and learning". Proceedings of the IEEE, v. 78(9), 1990, pgs. 1481-1497.

[33]Pineda, F. J., "Recurrent Backpropagation and the Dynamical Approach to Adaptive Neural Computation". Neural Computation, v. 1, 1989, pgs. 161-172.

[34]Cybenko, G. "Approximations by Superpositions of a Sigmoidal Function", Mathematics of Control, Signals and Systems, v. 2(4), 1989, pgs. 303-314.

It is this ability to form closed prototypes with a single hidden node that allows the projection network to be initialized rapidly to a good starting point which is already close to a desirable error minimum. Any of a number of algorithms can be used for this initialization; Kohonen learning, RCE, and ART are examples. Or prototypes may be placed around each training point. Once the network has been initialized in this manner, a modified backpropagation training, which will be discussed in Section 2.2, is used to adjust the network weights and thresholds to ensure error minimization. Because the network begins near a good solution, one avoids the long training time which standard backpropagation would take to reach this point as well as the possibility of getting stuck in local minima which might prevent one from reaching this point. As demonstrated in Section 3, this may lead to orders of magnitude reduction in training time.

The projection network has the added attraction of modularity: One network can be trained to recognize inputs of a set of classes and another can be trained to recognize inputs of other classes or different members of the same classes. Then, the two networks can be combined into a single network. Similarly, for function fitting applications, a network trained to output the value of a function over some range of inputs can be combined with one trained over a different range. In general, some additional training after the combination will be required to optimize the network performance, but there is no need to completely retrain the combined network, as would be generally required by conventional BPNN. This modularity is possible primarily because there is little or no interference between prototypes on the intermediate layer, particularly between prototypes of opposite classes, so that the addition of more prototypes does not necessarily destroy the signal to the output nodes.

The extension of a standard neural network to produce the projection network is a very simple one. The neural network inputs are projected onto a hypersphere in one higher dimension and the input and weight vectors are confined to lie on this hypersphere. A single hidden level node is now capable of forming either an open or a closed region in the original input space, as will be shown in the following section. This basic concept is not new. The need to normalize the input vector and the weight vector so that their dot product is a measure of their closeness has been recognized for a long time. Telfer and Casasent[35] have used a projection onto a cylindrical hyperbola for initialization of a network with no hidden layers. Saffrey and Thornton[36] have applied stereographic projection to the Upstart algorithm. What has not been recognized is that by projecting the input vector onto a hypersphere in one higher dimension one can create prototype nodes with closed or open classification surfaces all within the framework of a backpropagation-trained feedforward neural network. In this way one achieves rapid prototype formation through initialization and subsequent optimization through BP training. Furthermore, in contrast to the projection used by Telfer and Casasent, the projections illustrated in this paper allow formation of arbitrary size and position of the prototypes.

---

[35]Telfer, B. & Casasent, D. "Minimum-cost Ho-Kashyap Associative Processor for Piecewise-Hyperspherical Classification". IJCNN 1992 II, pgs. 89-94.

[36] Saffery, J. & Thornton, C. "Using Stereographic Projection as a Preprocessing Technique for Upstart". IJCNN 1992 II., 1991, pgs. 441-446.

2. Theory

2.1 The Projection Neural Network

To construct the projection neural network, we project each N-dimensional input vector $\boldsymbol{x}$ onto an $(N+1)$-dimensional input vector $\boldsymbol{x}'$ subject to the constraint that $|\boldsymbol{x}'| = R$. In other words, the new input vector $\boldsymbol{x}'$ is confined to lie on an $(N+1)$-dimensional hypersphere of radius R, and is the "projection" of $\boldsymbol{x}$ onto this sphere. Primed vectors refer to the (N+1)-dimensional projection of the unprimed, N-dimensional vectors. The projected inputs serve as the inputs into a standard feedforward neural network with an additional node in the input layer. There are a number of such projections, one example of which is a perspective projection defined by

$$\boldsymbol{x}' = R\left(\frac{h}{\sqrt{h^2+\boldsymbol{x}^2}}, \frac{\boldsymbol{x}}{\sqrt{h^2+\boldsymbol{x}^2}}\right), \tag{2.1}$$

where h is the distance between the origins of the N-dimensional space and the $(N+1)$-dimensional space and is a free parameter. R and h must be chosen in such a way so that the inputs will project onto a good portion of the hypersphere and can be easily separated. In other words, one must avoid projecting all the inputs into too small a portion of the hypersphere. The first component of $\boldsymbol{x}'$ in 2.1 is the component of the projected vector along the extra dimension. The remaining components lie in the original N-dimensional space. Since $\boldsymbol{x}'$ is an $(N+1)$-dimensional vector, the weight vector $\boldsymbol{w}'$ which connects it to any node in the first intermediate layer must also be an $(N+1)$-dimensional vector. In addition, $\boldsymbol{w}'$ is also forced to lie on the $(N+1)$-dimensional hypersphere so that its magnitude is always equal to the radius R :

$$|\boldsymbol{w}'| = R. \tag{2.2}$$

We illustrate this projection for 2-D inputs projected onto a 3-D sphere in Figure 1. The components of $\boldsymbol{x}'$ given above can be easily derived from Figure 1 using similar triangles and the Pythagorean theorem. This projection describes a mapping from the plane to a sphere which is carried out by drawing the line determined by the origin (the center of the sphere) and any point on the plane. The intersection of this line with the sphere is the projection of that point onto the sphere. The projection from the sphere back onto the plane maps circles on the sphere onto conic sections (circles, ellipses, hyperbola or lines) on the plane, depending on the size and location of the circle. We will use this perspective projection in the following discussion and the applications.

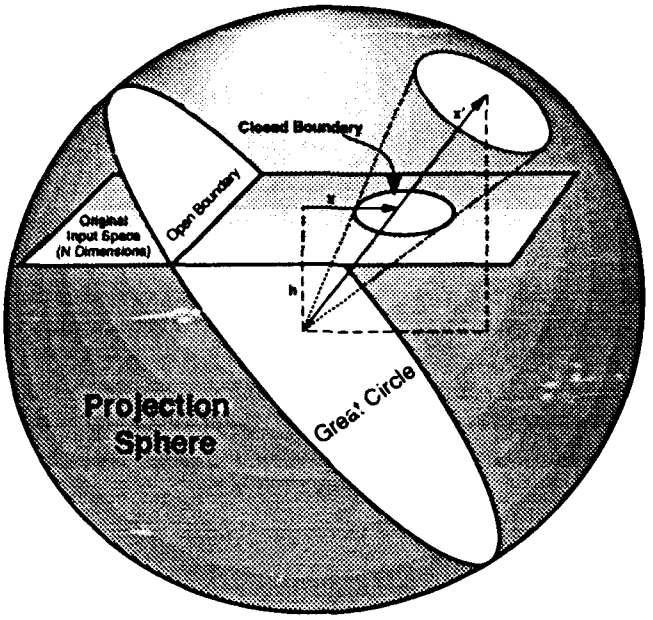


Figure 1. The projection transformation and the formation of boundary surfaces

$\boldsymbol{x}$ is the input vector in the original N-D space and $\boldsymbol{x}'$ is the input vector in the projected $(N+1)$-D space and lies on the hypersphere. Note that for large v the decision boundary is a circle on this sphere, and its projection back onto the plane is an ellipse. For v=0, the decision boundary is a great circle and its projection back onto the plane is a line.

The reason for adding the extra dimension to the inputs before normalization is to preserve all the information contained in the input vector, particularly its overall magnitude. In contrast, a simple normalization of $\boldsymbol{x}$ would confine the inputs to a hypersphere, but it would lose potentially valuable information contained in the magnitude of each input. This can be important if the radial direction contains important discriminatory information. In addition, such a scheme would not allow sufficient flexibility in the choice of the shape of a prototype's decision surface.

An alternate projection, described by the following formula, maps circles on the sphere to circles on the plane for the example of 2-D inputs:

$$\boldsymbol{x}' = R\left(\frac{1-(x/2h)^2}{1+(x/2h)^2}, \frac{x/h}{1+(x/2h)^2}\right). \tag{2.3}$$

When h is chosen to be R/2 this becomes the well known stereographic projection.

To demonstrate how projecting the input and weight vectors to $N+1$ dimensions allows us to draw hyperspheres or hyperplanes as decision boundaries, we note that the input to any intermediate level node is

$$w' \cdot x' - n \tag{2.4}$$

where v is the threshold for that node. The output of the node is a nonlinear function of this argument. It is common to use the so-called sigmoidal function: $\sigma(\xi) = \frac{1}{1+e^{-\xi}}$, which monotonically increases from 0 when the argument is $-\infty$ to 1 when the argument is $+\infty$.

Because σ provides a one-to-one mapping, the nodal outputs have a constant value when the nodal inputs have a constant value. For example, an intermediate node has an activation value of 1/2 when the input to that node is 0. For a given nodal activation value, say 1/2, there is a corresponding surface in the input space which is an effective decision surface; input vectors which lie on one side of this surface will produce activations less than 1/2 and input vectors on the other side will produce activations greater than 1/2. In the original input space, the surface is a hyperplane described by

$$w \cdot x - v = constant. \tag{2.5}$$

The constant is 0 when the decision surface is chosen to correspond to an activation of 1/2 . For a given activation level and hence a given value of the constant, the threshold determines the location of the hyperplane. With the constant chosen to be 0, the threshold is proportional to the distance of the hyperplane from the origin.

In N+1 dimensions both w' and x' lie on a hypersphere of radius R, and therefore the decision surface is described by

$$w' \cdot x' - v = R^2 \cos\theta - v = 0, \tag{2.6}$$

which is the equation of a hypersphere in N-dimensions (a circle on the 3-D sphere shown in Fig.1). The geometrical shape of its projection back onto the N-dimensional input space is determined by the choice of v: It is an ellipsoid or hypersphere if the surface is contained within the range of the N-dimensional input space and it is a hyperboloid or hyperplane if the surface extends beyond the limits of this space as shown in Fig. 1. If $v = 0$, $w' \cdot x' = 0$; the weight vector is perpendicular to the input vector, and the decision surface is a great circle of radius R on the $(N+1)$-dimensional hypersphere (see Fig. 1). Its projection back onto the N-dimensional space is an $(N-1)$-dimensional hyperplane (also shown in Fig. 1). If $|v/R| > R$, no solution exists on the hypersphere. At $v/R = R$, the surface is a point and all points in space lie outside or on the boundary of the decision surface, and at $v/R = -R$, all points lie on or inside the decision surface.

Thus, the projection network has the option to choose and the ability to use either hyperplanes or hyperspherical prototypes or both for the hidden nodes as is deemed convenient or conducive to the solution of any particular problem, whereas standard classifiers use one or the other (e.g. standard BPNN uses only hyperplanes and RCE uses only hyperspheres).

2.2. Initialization of Weights and Thresholds

Next, we will show that the projection operation allows us to make a good guess of the initial settings of the weights and thresholds. We note from equation 2.5 that when the inputs and weights are confined to the $(N+1)$-dimensional hypersphere, the maximum possible value of the input to any hidden layer node occurs when $w' = x'$ for any given threshold value. Therefore, if the weight vector w' of a node in the hidden layer is set equal to some input x' of class c,

$$w' = x' , \tag{2.7}$$

then the output of that node will be a maximum when the input is x'. If the decision boundary is chosen to be a hypersphere, then that node becomes a prototype hypersphere of class c with a radius v, designed to fire maximally when x' is the input. The radius v of the prototype can be set so that its projection back onto the original space is some desired closed or open region, and this initialization is often crucial in reducing the training time of the network. Examples of such initializations of the thresholds are shown in Appendix A.

With a sufficient number of such hyperspherical prototypes in the hidden layer corresponding to representative examples of input vectors of each class, the input space can be covered and classified with hyperspheres in a manner similar to an RCE network. Clearly, this initial setting of the weight vectors is much better than a random guess and can be very close to a desirable minimum or the error. The subsequent training process which corresponds to the translation, expansion and shrinking of hyperspheres will further reduce the error, and through the modified BP gradient descent technique discussed in section 2.3, the error in the outputs can be minimized.

If the desired decision surface is a hyperplane in the original input space, $(v = 0)$, then the weight vectors can be initially set so that the hyperplane crosses the input point. This ensures that the decision boundary will lie in an appropriate part of the original input space and is far more likely to be near the optimum position and orientation than a randomly chosen hyperplane, particularly in a high dimensional space.

So far, we have only applied the projection to the input layer nodes and set the first layer weights and thresholds. However, the projection operation need not be restricted to the input layer alone; it can be applied to any layer of the network by adding an extra node to that layer and transforming the nodal values so that they correspond to a projection onto a hypersphere in one higher dimension. Thus, the projection procedure can be repeated on any hidden layer of a multi-layer network in exactly the same manner as performed on the input layer. As an example application, for image recognition problems the first layer of hidden nodes in a network may be trained to form prototypes which recognize simple features such as edges or lines, and the next hidden layer of nodes may be trained to form prototypes of simple images based on the lower level features. The weights and thresholds of every projected layer can be initialized in the same manner as the first layer and they are trained with the modified backpropagation algorithm discussed in section 2.3.

If an appropriate set of prototypes are formed on the last hidden layer of the network, then the weights from this layer to the output layer and the thresholds of the output layer should also be initialized in order to reduce the training time. Fortunately, it is usually easy to construct a good set of initial values for the highest level weights and thresholds in a manner similar to RCE networks[2]. For classification problems, each output node representing a particular class will be connected to all the prototype nodes of that class in the hidden layer with a weight of +1 (or some other positive value), and it will be connected to prototype nodes of other classes with a zero or nearly zero weight. (Sometimes it may be desirable to use a negative weight for prototypes of other classes to suppress their contribution.) In general, the thresholds for each output class node should be set high enough to be above the level received by each node when inputs of the other classes are introduced. Typically, this is just above zero, but in some cases, especially if there is much overlap between classes, the thresholds may need to be higher. Similarly, for a function fitting problem, the weights can be set to the value of the function at the position of the corresponding prototype while the thresholds are set to 0 and the sigmoidal function can be omitted for the output nodes. With these initial settings of the weights and thresholds, the network training time can be reduced by orders of magnitude over training times of conventional BPNN as we will demonstrate for selected applications in section 3.

2.3. Training the Projected Weights; Gradient Descent on the Hypersphere

The backpropagation training algorithm minimizes the error in the output vectors by moving each weight vector in the direction of maximum error decrease, that is the direction opposite the gradient of the error with respect to the weights. However, in order to keep the weight vectors on the hypersphere, the standard backpropagation training algorithm must be modified: the weight vectors must be moved in the direction of maximum error decrease tangent to the hypersphere surface:

$$\delta w' = \frac{w'}{R} \times \left(\frac{w'}{R} \times \alpha \nabla \varepsilon \right), \tag{2.8}$$

where ε is the error in the outputs (the mean squared difference of network outputs and desired outputs, for example), $\nabla\varepsilon$ is the gradient with respect to the weight vector w, and α is the gain. However, since finite steps must be taken on any computer, this δw will slightly increase the radius of the weight vector. In order to prevent repeated iterations from moving the vectors off the hypersphere, the weights must be normalized to have a magnitude R, which is accomplished by the following replacement:

$$w' \rightarrow R \frac{w'+\delta w'}{|w'+\delta w'|} . \tag{2.9}$$

Although the above prescription is intuitively clear, a more mathematically rigorous treatment is provided by expressing the weight vector w' in a form that automatically guarantees a constant magnitude by the introduction of an unconstrained weight vector W':

$$w' = R\frac{W'}{|W'|}, \quad \text{and thus} \quad |w'| = R. \tag{2.10}$$

Minimizing the network output error with respect to the new unconstrained weight vector W' reproduces the results in Equations 2.9 and 2.10. Note that the threshold v is trained using standard backpropagation.

3. Applications

3.1 Simple 2-Dimensional Classification Problem

To demonstrate the advantages of the projection network, we apply it to selected problems. The first example is a simple two-dimensional problem which illustrates two advantages of this network, namely its ability to reduce the required number of hidden layer nodes by using hyperspheres or hyperplanes and its ability to reduce the training time through a good initialization of the weights and thresholds. The problem consists of two classes of points as shown in Figure 2, where the class 1 points are in the shaded regions and the class 2 points are in the white region. A hyperplane classifier such as BPNN will need three lines to enclose the circle and an additional line to separate the region at the right, whereas a hypersphere classifier may need several circles to define the linear boundary especially if the circles are not allowed to expand as in an RCE network. It will need only one circular prototype to enclose the circle provided that the prototype is allowed to shrink and expand during training; otherwise it will again need more than one prototype. The projection network needs only two prototypes and therefore only two intermediate layer nodes to classify the circular and rectangular regions.

A standard BPNN with 4 hidden layer nodes is trained on this data, with 2 input nodes corresponding to the x and y coordinates of the input point, and a single output node which gives the class of the input point. At the beginning of the training process, the BPNN tries to classify the inputs with just a single hyperplane as shown in Fig. 2a. Between 5,000 and 50,000 trials it adjusts this single hyperplane for optimal results and the fraction misclassified varies from 40% to 50%. This is the first local minimum which delays the training process. At 55,000 trials, the network brings in a second hyperplane as shown in Figure 2b, and between 55,000 and 90,000 trials it adjusts both hyperplanes to obtain the best results. This corresponds to the second local minimum visited by the network. At 95,000 trials, a third hyperplane and shortly thereafter the fourth hyperplane is engaged and adjusted until the final solution is reached as shown in Fig. 2c. The error now drops from 27% to 5%. This tendency of BPNN to attempt to solve a problem by sequentially engaging one hyperplane at a time is largely responsible for the delays caused by local minima for this problem and for large classes of problems.

However, the projected network can immediately engage all the hidden layer nodes as prototypes. To demonstrate this, the projected network is trained on the same data with only 2 hidden layer nodes. Its initial solution already uses both hyperspheres

(two circles) since they are ini ially set equal to input points chosen at random as shown in Fig. 2d. The initial fraction misclassified without training is 26.8%. Between 0 and 10,000 trials, it adjusts these two prototypes (see Fig. 2e); one circle is expanded to match the circular gray region, and the other is expanded until it approaches a line coinciding with the linear boundary (v approaches zero). At 10,000 trials, the decision boundaries closely match the class boundaries, and the fraction of misclassified points drops to 5.6% (see Fig. 2f).

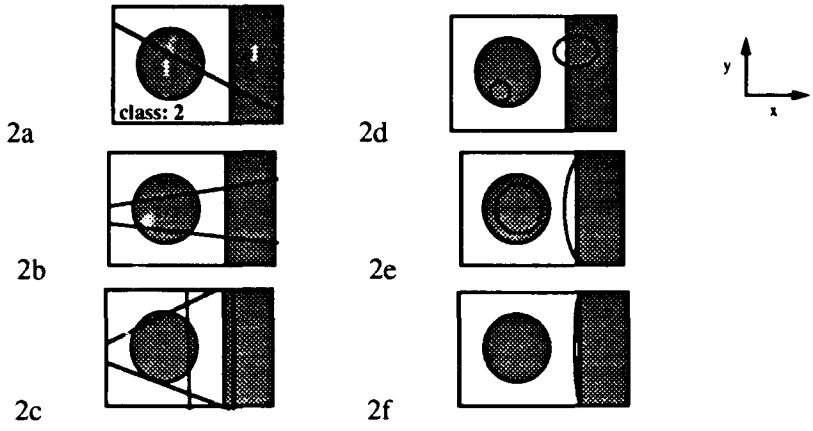


Figure 2. Illustration of formation of neural network solution for a two-dimensional problem. 2a. t=5000. 2b. t=55,000. 2c. t=95,000. Backpropagation-trained network learns by bringing in one hyperplane at a time. 2d. t=0. 2e. t=1,000. 2f. t=10,000. Projection network initially sets all prototypes and rapidly optimizes their position and size.

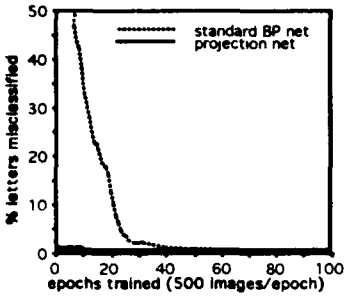
3.2 Optical Character Recognition

To illustrate these same advantages of the projection network on a more practical problem and to demonstrate its modular nature, we apply it to a character recognition problem which consists of the 26 letters of the alphabet on a 7 by 10 grid of pixels with gray scale values ranging from -0.5 to 0.5. Each character is allowed to translate by plus or minus one pixel in the x direction for a total of 3 possible positions of each character on the grid. Also, to each pixel gray value we add noise of amplitude + or -0.7 times a random number between 0 and 1, and then truncate back to the range -0.5 to 0.5 if the addition of noise takes the pixel value beyond this range. As shown in Fig. 3a below, these translations, coupled with the high level of noise, make the characters difficult to recognize even with the human eye.

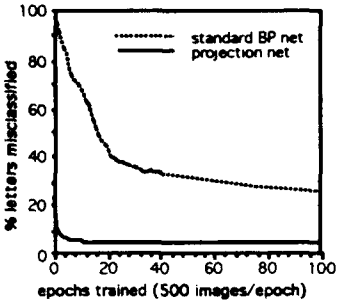
Example Characters with Noise:

3a.

A B C



3b.



3c.

Fig. 3. Comparison of backpropagation and projection networks for character recognition.
3a. Example characters. 3b. Performance on test set with no translation.
3c. Performance on test set with ± 1 pixel translation along x.

As shown in Figures 3b and 3c the slow training time for the standard backpropagation trained net is greatly reduced by the projection net which begins at a good solution after one pass through the training data set (indicated as epoch 0 on the graphs). The standard net had 70 input nodes, 50 intermediate level nodes, and 26 output nodes for the results shown in Figure 3b in which the characters are not translated. With translation allowed the number of intermediate level nodes was increased to 100. The corresponding projection networks simply had one additional input node.

The advantage of the modular nature of the projection network is demonstrated by combining two separately trained networks. We trained one network with 124 intermediate level nodes and 12 output nodes to recognize the letters A through L and another network with 126 intermediate level nodes and 14 output nodes to recognize M through Z. Translations of ±1 pixel along both x and y were allowed. After 5,000 trials, the first network misclassified 3.4% of the characters and the second network misclassified 1.2% of the characters. After combining and without training, the average misclassification error was 6.9%. After 6,000 training trials, the fraction missed dropped to 3.1%, and after 13,000 trials, it was 2.9%. The important result of this experiment is that the combined network showed a low initial error, demonstrating that the projection network can be nearly modular, and that the combination of networks in real applications is practical.

3.3 Handwritten Word Recognition

To demonstrate the ability of the projection network to rapidly solve large scale problems, we applied it to handwritten word recognition. A sample of the handwriting used is shown in Figure 4. The inputs consisted of thirty images of handwritten words written by the same individual and digitized into 60 x 340 pixels. Thus the projected

neural net had 20,400 + 1 input nodes and 30 output nodes corresponding to the 30 different words. With no noise added, the initial setting of the weights and thresholds is sufficient to classify all 30 training images perfectly without additional training. This is shown in Figure 5 which contrasts this single pass initialization with the slowness of a standard BPNN. The projection network memorizes all 30 words after a single pass through the training set. This is a remarkable performance for such a large dimensional problem. Note that this example is not meant to demonstrate a network capable of word recognition. Indeed, such a network generalizes poorly on images it has not trained on. No attempt was made to train the network to be insensitive to translation, scale changes or local distortions which are typical of handwriting. Such insensitivity is best built in by analyzing the image with local overlapping regions each of which is trained to be insensitive to small distortions. What this experiment does demonstrate, however, is the feasibility of rapidly training a neural network to learn a problem of large dimensions.

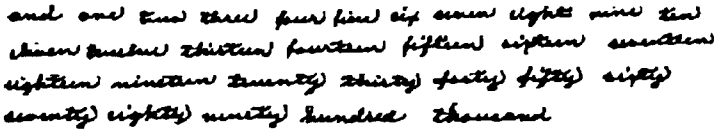


Figure 4. **Training samples of handwritten words.**

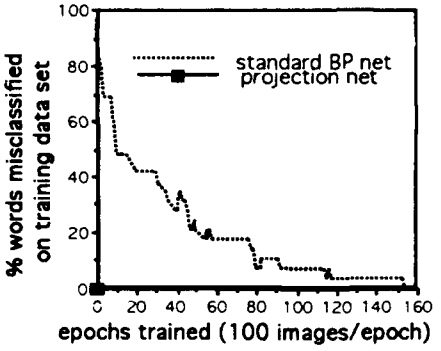


Figure 5. Comparison of backpropagation vs. projection network training for handwritten word identification. Results refer to the training data only.

4. Summary

We have developed a new neural network by projecting the input vector of a standard feedforward neural network onto a hypersphere in one higher dimension. The network is trained with a modification of backpropagation algorithm. We have demonstrated through theoretical discussions and practical applications that the network has several advantages over traditional BPNN: By allowing a good initial setting of the weights and thresholds, the projection network can save orders of magnitude in the

training time. It is more efficient in its use of intermediate layer nodes, since it can use either hyperspheres or hyperplanes as decision boundaries wherever appropriate. It is also modular in many applications; two or more independently trained networks can be combined and then trained for a relatively short time with good results. Finally, the projection operation is not restricted to the input layer but can be performed on any layer of a multi-layer network as required, thus further enhancing the network capability and decreasing the learning time.

5. Appendix

Shapes of Classification Regions and Network Initialization

The projection of the hyperspherical decision boundary on the *(N+1)*-dimensional hypersphere back onto the original *N*-dimensional space can produce open or closed quadratic surfaces as shown in Fig. 1 for N=2. To show this, begin with the projection defined by

$$\boldsymbol{x}' = \left(x'_0 \, , \; \boldsymbol{x}'_\perp\right) = R\left(\frac{h}{\sqrt{h^2+\boldsymbol{x}^2}} \, , \; \frac{\boldsymbol{x}}{\sqrt{h^2+\boldsymbol{x}^2}}\right), \qquad \text{(A.1)}$$

where x'_0 and $\boldsymbol{x}'_\perp$ are components of the projected vector along the additional axis, labeled '*0*', and the original space respectively. Similarly, the *(N+1)*-dimensional weight vector $\boldsymbol{w}'$ can be expressed in terms of w'_0 and $\boldsymbol{w}'_\perp$:

$$\boldsymbol{w}' = \left(w'_0 \, , \; \boldsymbol{w}'_\perp\right) \; .$$
(A.2)

The activation of a hidden level node is a nonlinear function of the inner product of the projected input vector with the nodal weight vector and is thus a constant when the inner product is constant. If this constant is set to zero, then the decision surface of any hidden node is defined by

$$\boldsymbol{w}' \cdot \boldsymbol{x}' - v = 0 \; , \quad \text{which implies} \qquad w'_0 h + \boldsymbol{w}'_\perp \cdot \boldsymbol{x} = \frac{v}{R}\sqrt{h^2+\boldsymbol{x}^2}. \qquad \text{(A.3)}$$

Since the choice of coordinates in the *N*-dimensional space is arbitrary, we can simplify this expression by choosing x_1 along $\boldsymbol{w}'_\perp$ without any loss of generality:

$$w'_0 h + w'_\perp x_1 = \frac{v}{R}\sqrt{h^2+\boldsymbol{x}^2}, \text{ where } \boldsymbol{x}^2 = \boldsymbol{x}_\perp{}^2 + x_1{}^2 \text{ , and } w'_\perp = |\boldsymbol{w}'_\perp|. \qquad \text{(A.4)}$$

Here $x_\perp$ is the projection of x in a direction orthogonal to the first axis and should not to be confused with $\boldsymbol{x}'_\perp$ defined in A.1. This can be cast into the standard form for the equation of an ellipse:

$$\frac{x_\perp^2}{b^2} + \frac{(x_1 - x_c)^2}{a^2} = 1, \qquad \text{(A.5)}$$

where a and b are the major and minor axes of the ellipse respectively, and are given by

$$b = h\sqrt{\frac{(R^2/\nu)^2 - 1}{1 - (w_\perp R/\nu)^2}}, \quad a = \frac{b}{\sqrt{1 - (w_\perp R/\nu)^2}}, \qquad \text{(A.6)}$$

and the center of the ellipse is located along the first axis with the coordinate value

$$x_c = \frac{w_0 w_\perp R^2 h/\nu^2}{1 - (w_\perp R/\nu)^2}. \qquad \text{(A.7)}$$

As long as the quantities within the square roots are positive, the equation describes a hyperellipse (a closed surface). If one of the quantities is negative the equation describes an open hyperbolic surface. In the special case in which the threshold is 0 the equation is that of a hyperplane. If one desires to set a prototype node whose decision boundary is an ellipse centered around an input point $\boldsymbol{x}$ with a minor axis of b, then the weight vector is set as

$$w'/R = \left(\frac{(b^2+h^2)/h}{\sqrt{x^2 + (b^2+h^2)^2/h^2}}, \frac{x}{\sqrt{x^2 + (b^2+h^2)^2/h^2}} \right), \qquad \text{(A.8)}$$

and the threshold is set equal to

$$\nu = R^2 \sqrt{\frac{x^2 + b^2 + h^2}{x^2 + (b^2+h^2)^2/h^2}}, \qquad \text{(A.9)}$$

which is obtained from equations A.6 and A.7. Alternatively, the weight vector can be set equal to the projected input vector to describe a prototype which responds maximally to this input point. The center of the ellipse, $\boldsymbol{x}_c$, is then offset from the input point:

$$\boldsymbol{x}_c = \boldsymbol{x}\,(1 + b^2/h^2). \qquad \text{(A.10)}$$